ODLANIGER LOURENÇO DAMACENO MONTEIRO

APLICAÇÃO DE PADRÕES DE PROJETO NO DESENVOLVIMENTO DE FRAMEWORKS - UM ESTUDO DE CASO

Florianópolis - SC

2002

UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Odlaniger Lourenço Damaceno Monteiro

APLICAÇÃO DE PADRÕES DE PROJETO NO DESENVOLVIMENTO DE FRAMEWORKS - UM ESTUDO DE CASO

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Orientador:

Prof. Rosvelter João Coelho da Costa

Florianópolis, 08 de agosto de 2002.

APLICAÇÃO DE PADRÕES DE PROJETO NO DESENVOLVIMENTO DE FRAMEWORKS - UM ESTUDO DE CASO

Odlaniger Lourenço Damaceno Monteiro

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação com área de concentração em Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Prof. Dr. Fernando Álvaro Ostuni Gauthier
Coordenador do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina

Banca Examinadora:

Prof. Dr. Rosvelter João Coelho da Costa
Orientador e Presidente da Banca
UFSC

Prof. Dr. João Bosco Mangueira Sobral
UFSC

Prof. Dr. Vitório Bruno Mazzola
UFSC

Agradecimentos

Agradeço ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina pela oportunidade de ter me recebido como aluno.

Agradeço ao Centro Universitário do Pará pela oportunidade que me foi dada de aderir, seguir a carreira acadêmica e viabilizar a minha participação no programa de Pós-Graduação em Ciência da Computação em parceria com a Universidade Federal de Santa Catarina.

Agradeço ao meu orientador Prof. Rosvelter João Coelho da Costa, pelo direcionamento e grande apoio na elaboração deste trabalho, bem como pela paciência e dedicação oferecida.

Agradeço aos meus colegas professores, dos quais muitos participaram do mesmo programa de Pós-Graduação, pela amizade e troca de colaborações durante o período de curso.

Agradeço ao professor Gustavo Campos, responsável pela coordenação local (Belém) do programa de Pós-Graduação pela amizade e suporte durante o transcorrer do curso.

Agradeço especialmente à professora Conceição Rangel Fiuza de Melo, pelo incentivo diário e apoio irrestrito para a elaboração desta pesquisa e conclusão do curso.

Agradeço à minha família, meus irmãos Filipe e Davi e meus pais Reginaldo e Iara Monteiro, por formarem a estrutura abençoada que me foi dado o privilégio de integrar.

Agradeço à minha família, minha esposa Lucélia e meu filho Leandro, pelo incentivo, amor e carinho e por nortearem minhas principais atitudes, tanto pessoais quanto profissionais.

Agradeço a Deus por sempre me conduzir por bons caminhos.

Sumário

Lista de figuras

Resumo

Este trabalho aborda o projeto de sistemas orientados a objetos auxiliado por padrões de projeto. Fornece uma visão geral sobre os principais aspectos da modelagem de sistemas orientados a objetos, conceitos, notações, padrões de projeto e frameworks, e suas aplicações no processo de desenvolvimento de software.

O principal foco desta dissertação é o uso de padrões de projeto na definição de arquiteturas de software oferecendo alto grau de reusabilidade. É apresentado o desenvolvimento de um sistema avaliador de expressões matemáticas – estudo de caso – onde os padrões de projetos foram utilizados para a solução de diferentes problemas de projeto.

Um protótipo totalmente funcional desse sistema foi desenvolvido utilizando a plataforma Java disponibilizada pela *SUN Microsystems.*

Palavras-chave: padrões de projeto; programação orientada a objetos; frameworks.

Abstract

This work is concerned with object-oriented software development based on design patterns. It will be shown an overview about the most important aspects of object-oriented software modeling, concepts, notation, design patterns, frameworks and their application on software developing process.

The main point of this work is the use of design patterns in software architecture definition with high level of reusability. It is presented a mathematics expression evaluator framework, where design patterns were used to solve different kind of reusability problems.

A totally functional prototype of this solution was developed using *SUN Microsystems* Java platform.

Key-words: design patterns; object-oriented programing; frameworks.

Capítulo 1

Introdução

Projetar software orientado a objetos é uma tarefa difícil, mas projetar software reutilizável orientado a objetos é ainda mais difícil. É necessário modelar objetos pertinentes, fatorá-los em classes no nível correto de granularidade, definir as interfaces das classes e as hierarquias de herança e estabelecer as relações entre eles. O projeto deve ser específico para resolver o problema, mas também genérico o suficiente para atender futuros problemas e requisitos [GAM2000].

A modelagem de uma arquitetura de sistema que tenha múltiplos requisitos e posteriores adaptações com modificações desses requisitos é um dos maiores desafios do arquiteto de sistemas. Ao mesmo tempo em que o software deve ser específico o suficiente para resolver um problema específico, ele deve também se adaptar a soluções um pouco mais genéricas.

Um professor de disciplina de programação de sistemas orientados a objetos pode perceber que apresentação a um aluno de graduação em computação de um conjunto de regras e padrões, previamente testados e aprovados, associado com conceitos de componentização, pode melhorar o tempo que um aluno leva para desenvolver uma boa solução de programa, com características de reusabilidade e adaptabilidade.

O principal problema que motivou a elaboração deste texto é a dificuldade de se desenvolver software, mesmo para principiantes em programação, com características de reusabilidade.

O tema principal tratado nesta pesquisa é a exploração dos elementos inerentes ao desenvolvimento de software reutilizável com soluções baseadas em padrões de projeto.

Para tal entendimento torna-se necessário elaborar um estudo sobre modelagem e desenvolvimento de software; padrões de projeto e suas aplicações.

A partir dessas idéias surgem dois questionamentos que norteiam este trabalho:

- Como os padrões de projeto podem nos ser úteis no processo de criação de sistemas orientados a objetos?
- O que é possível obter de vantagens ao optarmos por usar padrões de projeto no desenvolvimento de uma arquitetura de software?

O objetivo principal deste texto é estudar e mostrar a aplicação de padrões de projeto na realização de software reutilizável orientado a objetos.

Os objetivos específicos deste estudo são:

- Apresentar as características mais importantes da modelagem de software orientado a objetos, como complexidade e modelos de software;
- Visualizar os itens necessários para representar modelos de software orientado a objetos em uma linguagem de modelagem unificada UML;
- Conhecer os conceitos e aplicações dos principais padrões de projeto catalogados;
- Apresentar um exemplo de aplicação dos padrões de projeto na criação de um framework para solucionar um problema.

Para atender os objetivos propostos, esse trabalho está estruturado com o Capítulo 1 apresentando as motivações e a justificativa para o desenvolvimento desse trabalho, seus objetivos e estrutura.

O Capítulo 2 mostra as principais características da modelagem baseada em orientação a objetos, desde os princípios de complexidade de software, a importância do processo de modelagem, os modelos de software e discorre sobre os conceitos mais relevantes de uma linguagem de modelagem unificada, a UML.

O Capítulo 3 caracteriza os padrões de projeto e frameworks, conceitos e aplicações dos principais padrões catalogados e mostra um exemplo simples de aplicação do padrão Estado no projeto de uma calculadora.

O Capítulo 4 apresenta quatro versões de um sistema avaliador de expressões, onde em cada versão é aplicado um padrão diferente. São feitas análises da aplicação dos padrões suas conseqüências e benefícios.

No Capítulo 5 são feitas as considerações finais e propostas de continuação deste trabalho.

Capítulo 2

A Modelagem de Sistemas Orientados a Objetos

A elaboração de programas abrange um conjunto de regras e passos que devem ser seguidos para que o resultado satisfaça os objetivos esperados. Além disso, especifidades como o ambiente, os usuários, o grau de precisão das respostas do programa, etc., devem sempre ser respeitados.

A programação orientada a objetos vem de encontro com essas necessidades provendo inúmeros elementos relacionados tanto na análise como ao projeto de sistemas de software. Este capítulo abrange de forma sucinta o problema da modelagem de sistemas de software envolvendo o paradigma da programação orientada a objetos.

1 - Complexidade de software

A complexidade dos sistemas de software advém de inúmeros fatores e é geralmente comparável a complexidade encontrada em sistemas de engenharia. São quatro os principais fatores [BOO1994]:

1) A complexidade do domínio do problema

Os problemas a serem resolvidos por um software envolvem elementos de grande complexidade, com requisitos difíceis de compreender e, por vezes, contraditórios. Além disso, existem os requisitos não funcionais que nem sempre aparecem explicitamente, mas que são importantes como: usabilidade, desempenho, custo, tempo de vida útil e reutilização. Boa parte da dificuldade em compreender os requisitos do software advém dos problemas de comunicação e compreensão entre os usuários do programa e seus analistas. Por último, os requisitos de um sistema podem e geralmente mudam no decorrer do tempo, em função de mudanças nas regras e necessidades do ambiente. Como a elaboração de software resulta em um investimento de capital, o seu desenvolvimento geralmente envolve uma prorrogação, chamada inconvenientemente de manutenção.

2) A dificuldade de gerenciamento do processo de desenvolvimento

Uma equipe de desenvolvimento de software precisa ser bem gerenciada, pois além de ter como tarefa fundamental manter o usuário protegido da complexidade do programa, a equipe vai utilizar durante todo o processo de criação uma variada quantidade de artifícios, tornando necessário uma boa sintonia, comunicação, unidade e integridade entre seus membros.

3) A flexibilidade possível através do software

É necessário, possível e perigoso o uso da abstração no processo de elaboração, pois ao mesmo tempo em que se precisa atender as especificações do problema, existem alguns padrões de desenvolvimento na indústria de software que ainda não são bem interpretados e utilizados pelas empresas.

4) A problemática da caracterização do comportamento de sistemas discretos

Os sistemas discretos, diferentemente dos sistemas contínuos, possuem a dificuldade natural de serem modelados, pois devem ser representados por valores discretos e devem ser elaborados de tal forma que procurem atender da maneira mais completa possível todos os estados de um sistema. Um sistema computacional, montado em um ambiente digital, é um sistema discreto que possui variáveis para representar o estado de seus valores. Como é difícil prever todos os possíveis estados de um sistema contínuo, é necessário trabalhar com níveis aceitáveis de aproximação e erro.

Ao projetar software de qualidade deve-se levar em consideração que a complexidade é um fator presente e contínuo em todo o processo. Obtém-se um melhor aproveitamento, através do uso de técnicas de abstração e criação de software, entre elas, algumas que serão apresentadas no decorrer deste capítulo.

2 - Modelagem

Pode-se afirmar que a meta de um bom produto de software é atender as necessidades dos usuários com qualidade e seu desenvolvimento deve ser feito de maneira previsível e com utilização eficiente de recursos.

Dentro dessa perspectiva, a modelagem aparece como parte central e fundamental de todas as atividades de desenvolvimento de software, pois a construção de modelos permite que a estrutura e o comportamento desejados do sistema possam ser divulgados e comunicados.

2.1 - Importância da modelagem

Um programa de computador simples e pequeno talvez não exija uma modelagem completa, pois a sua complexidade pode não ser tão grande a ponto de impedir o entendimento e a construção da solução. Entretanto, um software com maior grau de complexidade exige cuidados tais como levantamento de requisitos, planejamento, desenvolvimento e validação, pontos para os quais uma modelagem bem elaborada é fundamental.

A modelagem é uma técnica de engenharia aprovada e bem-aceita. A sua aplicação na engenharia de sistemas é também aconselhável, pois modelos são simplificações da realidade e nós os utilizamos para compreender melhor o sistema que estamos desenvolvendo. Ainda em cima de seus princípios, citamos quatro objetivos da modelagem [BOO2000]:

- Os modelos ajudam a visualizar o sistema como ele é ou como desejamos que seja;
- Os modelos permitem especificar a estrutura ou o comportamento de um sistema;
- Os modelos proporcionam um guia para a construção do sistema;
- Os modelos documentam as decisões tomadas.

Quanto maior a complexidade do software, maior será a necessidade da utilização da modelagem, pois construímos modelos de sistemas complexos para compreendê-los em sua totalidade.

A modelagem pode ser feita até mesmo de maneira informal, através de rascunhos ou resumos de modelos, porém a utilização de métodos formais permite uma padronização do modelo e a garantia de que algumas regras de desenvolvimento pré-estabelecidas sejam obedecidas.

2.2 - Princípios da modelagem

Baseado no estudo do uso da modelagem é possível afirmar quatro princípios norteadores [BOO2000]:

A escolha dos modelos a serem criados tem profunda influência sobre a maneira como um determinado problema é atacado e como uma solução é definida.

Cada modelo poderá ser expresso em diferentes níveis de precisão.

Os melhores modelos estão relacionados à realidade.

Nenhum modelo único é suficiente. Qualquer sistema não-trivial será melhor investigado por meio de um pequeno conjunto de modelos quase independentes.

3 - Modelos de software

Desde que o homem passou a usar máquinas de computação, surgiu a necessidade de repassar as instruções e os dados para a obtenção da solução esperada. Simultaneamente, de acordo com a evolução tecnológica, aumentou também o grau de dificuldade dos problemas propostos. Veremos a seguir um pouco do histórico da evolução das metodologias de desenvolvimento de sistemas e alguns dos conceitos básicos de uma técnica que vem sendo amplamente utilizada: a orientação a objetos.

3.1 - Foco em algoritmos

Durante muitos anos os computadores foram utilizados somente por grandes empresas. Até que no princípio da década de 70 houve uma queda no preço dos equipamentos de informática e algumas empresas de médio e pequeno porte puderam se aventurar em transferir para os sistemas informatizados algumas funções de caráter operacional.

Todo o conhecimento que se tinha até então de técnicas de desenvolvimento de software não era suficiente para contornar os problemas de desenvolvimento de sistemas, principalmente se produzidos em grande escala, como passou a se exigir com a demanda de um público consumidor de programas.

E desta necessidade surgiu uma técnica com o foco no desenvolvimento do algoritmo, que até hoje é bastante utilizada e difundida chamada de *programação estruturada*, seguida pelo conceito de *desenvolvimento estruturado de sistemas*. Como uma alternativa para sanar as dificuldades de um desenvolvimento em grande escala, a metodologia estruturada pregava alguns princípios [OLI1996]:

Abstração

A solução de um problema pode ser encontrada mais facilmente se o mesmo for analisado de forma a separar os demais aspectos que possam atrapalhar numa etapa (relevar os detalhes não necessariamente importantes);

Formalidade

Deve ser seguido um caminho rigoroso e metódico para solucionar um problema;

Dividir para conquistar

Dividir o problema em partes menores, independentes e com possibilidade de serem mais simples de entender e solucionar;

Hierarquização

Os componentes da solução devem ficar dispostos em uma estrutura hierárquica. O sistema deve ser entendido e construído nível a nível, onde cada novo nível acrescenta mais detalhes.

Como esses princípios facilitavam a vida dos programadores, a abordagem estruturada obteve grande sucesso e ainda hoje é amplamente utilizada.

3.2 - Foco em objetos

A revolução ocorrida na década de 70 voltou a ocorrer no final da década de 90, onde os preços dos equipamentos voltaram a cair. Um bom número de empresas, de diversos portes, já possuíam parte de seus sistemas com um considerável nível de informatização, foi amplamente divulgado o uso da Internet como meio de comunicação e busca maciça de informação e finalmente o computador passou a ser um

eletrodoméstico dos indivíduos de classe média e uma ferramenta de trabalho diário para uma grande quantidade de pessoas.

Surge então a necessidade de se produzir softwares mais atraentes, dinâmicos e com alto poder de troca de informações [OLI1996]. Tais aplicações se caracterizam por:

- Interação intensiva com o usuário;
- Uso de interfaces gráficas (GUI-Graphics User Interface);
- Necessidade permanente de alteração e expansão, dada a velocidade de mudanças na tecnologia do hardware;
- Interação com outros sistemas, possibilitando a troca de dados;
- Portabilidade para diversas plataformas e sistemas operacionais;

As técnicas oferecidas pela metodologia estruturada não foram suficientes para atender com a satisfação desejada a elaboração deste tipo de aplicação. Era necessário partir para outro tipo de metodologia, que permitisse o desenvolvimento de sistemas com essas novas características. Começou a ser adotada por parte dos profissionais da área de desenvolvimento de sistemas a *abordagem orientada a objetos*.

3.3 - Abordagem orientada a objetos

Apesar de não ser um conceito totalmente novo no meio acadêmico, somente na década de 90, esta abordagem força também no mercado de software. Basta citar que as grandes empresas da área de desenvolvimento de sistemas oferecem ferramentas baseadas no conceito de objetos.

A abordagem orientada a objetos é fundamentada no que coletivamente chamamos de modelo de objetos, que engloba os princípios da abstração, hierarquização, encapsulamento, classificação, modularização, relacionamento, simultaneidade e persistência. Embora não sejam individualmente novos, é importante ressaltar que no modelo de objetos esses elementos são agora agrupados de forma sinérgica.

Como a abstração e a hierarquização já foram detalhados na parte da abordagem estruturada, vamos analisar os outros princípios:

Encapsulamento

Mecanismo pelo qual podemos ocultar detalhes de uma estrutura complexa, que poderiam interferir durante o processo de análise. Por exemplo, todos nós sabemos que um carro é composto de motor, lataria, bancos, etc. O motor por sua vez é composto por uma grande quantidade de peças e elementos de ligações. Mas na hora de preencher um cadastro financeiro para solicitação de um empréstimo pessoal, nos campos para colocar os detalhes do carro, não será necessário dizer quais são as peças que formam o motor do carro, apesar de sabermos que sem estas, o valor do carro cai bastante. Estes detalhes serão encapsulados pelas características gerais do carro.

Classificação

É o ato de associar um objeto analisado a uma determinada categoria. Ao classificarmos um objeto, estamos afirmando que este pertence a uma determinada classe. Esta associação é feita comparando as características e funções do objeto em questão com as características e funções dos objetos que pertencem àquela categoria. Quando vemos um gato, por exemplo, podemos afirmar que ele pertence à classe dos mamíferos, porque conhecendo as características e funções dos mamíferos e dos gatos, fica fácil chegar a esta conclusão.

Modularização

Em um sistema previamente dividido, podemos juntar partes com algumas semelhanças. Note que a idéia de modularizar facilita bastante a aplicação dos outros princípios.

Relacionamento

Para o funcionamento do todo, é necessário que as partes funcionem separadamente, mas cooperativamente.

Paralelismo

Mesmo em um sistema simples, pode haver diversas ações a serem executadas concorrentemente. É necessário um correto gerenciamento dos recursos

computacionais para haver uma correta distribuição do tempo entre as tarefas a serem executadas.

Persistência

Um objeto em um programa utiliza algum espaço para armazenamento e manipulação e existe por um período de tempo em particular. Muitas vezes o tempo de vida de um objeto supera o tempo de vida do sistema que o manipula. Este princípio prega que deve haver uma atenção especial nesta manipulação.

3.3.1 - Objetos e classes

Usamos o termo objeto para representar um determinado elemento do mundo real. Mas somente analisaremos os objetos que têm relevância para a solução de um determinado problema. Portanto, o objeto é uma entidade do mundo real que merece representação para o ambiente estudado.

Como exemplos de objetos, podemos citar os objetos físicos (um livro, uma mercadoria), funções de pessoas para os sistemas (cliente, vendedor), eventos (uma compra, um telefonema), interações entre outros objetos (um item de uma nota fiscal é uma interação entre uma compra e um produto do estoque) e lugares (loja matriz, revenda norte).

Um objeto pode ser simples ou composto de demais objetos. Em geral a maioria dos objetos são compostos, pois sempre podemos dividí-los em partes menores até chegar a elementos indivisíveis. Um sistema é um grande objeto composto de outros objetos, formando um mecanismo.

Um objeto é composto de atributos e métodos e tem a capacidade de trocar mensagens com outros objetos.

Os atributos, ou propriedades, representam as características dos objetos e podem ser fixos ou variáveis. Por exemplo, o objeto computador tem como características a marca, o modelo, a quantidade de memória RAM, o tamanho do disco rígido, se tem ou não CD-ROM, etc.

O estado de um objeto é o conjunto de valores de seus atributos em um determinado instante. Os serviços ou métodos são as funções que operam sobre o

mesmo. O comportamento de um objeto é como ele age e reage em termos de suas mudanças de estado e troca de mensagens com outros objetos. A identidade é a característica que um objeto deve ter de ser distinguido dos demais.

Uma classe representa uma coleção de objetos que possuem características e comportamentos comuns e de agora em diante, diremos que um objeto é uma instância de uma determinada classe. Para estabelecer que criaremos nossos objetos baseados nas características definidas em suas classes. A ênfase da abordagem orientada a objetos é dada na descrição das classes, e não dos objetos, como se poderia pensar pelo nome. A Fig. 2.1 mostra um exemplo de classe e algumas de suas respectivas instâncias (objetos).

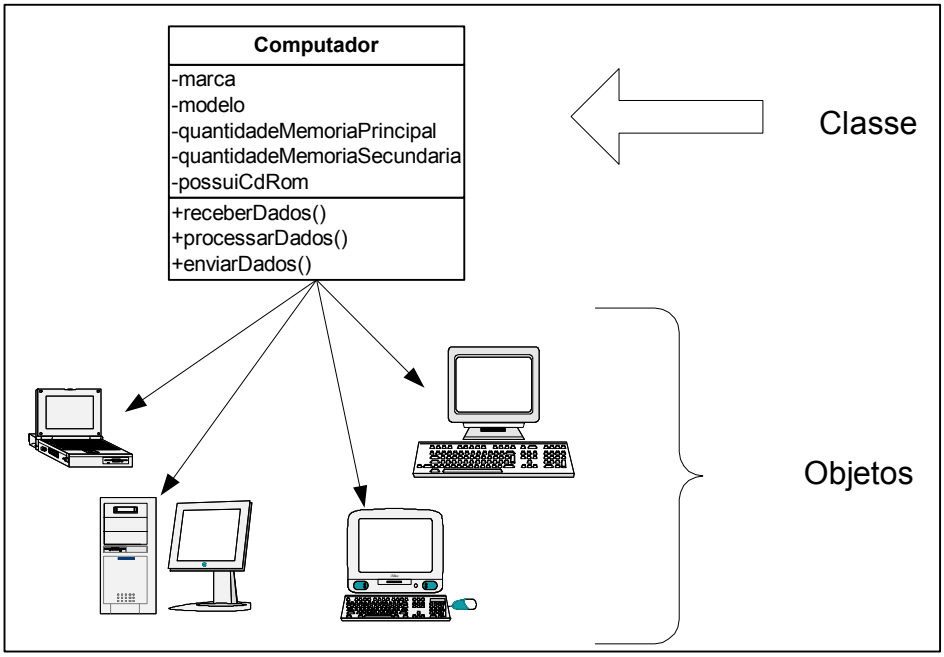


Figura 2.1 Exemplo de classe e suas instâncias (objetos).

Todo objeto pertence a uma determinada classe durante sua existência, não podendo modificar sua classificação.

3.3.2 - Relacionamento entre classes

As classes devem poder se relacionar para que o sistema possa funcionar. As principais formas de relacionamento são as seguintes:

Associação

Um exemplo típico de associação é a feita entre as classes Aluno e Responsável de um sistema de controle acadêmico. Um objeto da classe Aluno está associado a um único responsável, mas uma mesma pessoa pode ser responsável por mais de um aluno. Podem haver relacionamentos de cardinalidades um-para-um, um-para-muitos e muitos-para-muitos.

Especialização

Dada uma determinada classe, criamos uma outra com acréscimos ou modificações de novos atributos ou serviços que a tornam mais específica a um determinado comportamento.

Herança

É o mecanismo pelo qual uma classe obtém as características e métodos de outra para expandí-la ou especializá-la de alguma forma.

Agregação

É o ato de agregar, juntar duas ou mais classes para formar uma nova classe.

Além desses conceitos, se analisarmos as obras dos diversos autores que se propõem a estudar os detalhes da abordagem orientada a objetos, veremos uma quantidade muito grande de variações, conceitos, especificações e documentações.

3.4 - Desenvolvimento de sistemas orientados a objetos

É perfeitamente possível modelar uma solução utilizando totalmente a abordagem orientada a objetos desde a fase de análise, passando pelo projeto do software e chegando ao código através de uma linguagem de programação orientada a objetos.

Uma grande vantagem de se pensar totalmente orientado a objeto é o fato de que um mesmo objeto, concebido na fase de análise, passa com as mesmas características desde o usuário até o programador que será responsável pela codificação final.

A abordagem orientada a objetos teve diversas contribuições ao longo dos últimos anos. Entre elas os conceitos que Coad, Yourdon, Pressman e tantos outros abordaram, discutiram e definiram em suas publicações. Figurando entre os principais, temos:

- A orientação a objetos é uma abordagem para a produção de modelos que especifiquem o domínio do problema de um sistema.
- Quando construídos corretamente, sistemas orientados a objetos são flexíveis a mudanças, possuem estruturas bem conhecidas e provêm a oportunidade de criar e realizar componentes totalmente reutilizáveis.
- Modelos orientados a objetos são realizados convenientemente utilizando uma linguagem de programação orientada a objetos. A engenharia de software orientada a objetos é muito mais que utilizar mecanismos de sua linguagem de programação, é saber utilizar da melhor forma possível todas as técnicas da modelagem orientada a objetos.

A abordagem orientada a objetos não é só teoria, mas uma tecnologia de comprovada eficiência e qualidade usada em inúmeros projetos para construção de diferentes tipos de sistemas.

A abordagem orientada a objetos requer um método que integre o processo de desenvolvimento e a linguagem de modelagem com a construção de técnicas e ferramentas adequadas.

Um dos problemas de se tentar estudar a abordagem orientada a objetos é o fato de que alguns autores montam suas próprias concepções sobre esta teoria, surgindo assim várias escolas. A Linguagem de Modelagem Unificada – UML (“Unified Modeling Language”) surgiu como uma tentativa de solução a este problema e tenta concentrar os principais conceitos da modelagem orientada a objetos em um único método.

4 - Linguagem de Modelagem Unificada - UML

Para utilizarmos qualquer método no desenvolvimento de software, precisamos ter um conjunto de termos, nomenclaturas e ferramentas padronizadas formando uma linguagem ou notação que os técnicos entendam e efetivamente utilizem. A UML vem justamente de encontro a estes objetivos.

A UML foi desenvolvida por Grady Booch, James Rumbaugh, e Ivar Jacobson que são três conhecidos autores de metodologias de desenvolvimento de software seguindo a abordagem orientada a objetos. A UML nasceu da junção do que havia de melhor nas suas respectivas metodologias.

A seguinte definição aparece em [MAT2002]:

"A UML é uma composição das boas características de outras notações de ferramentas direcionadas à orientação a objetos, tornando-se assim a ferramenta ideal para conceber, compreender, testar, validar, arquitetar lógica e fisicamente e ainda identificar todos os possíveis comportamentos do sistema".

Para o exercício de sua função, a UML possui três grandes blocos para modelar um sistema: itens, relacionamentos e diagramas. Itens são as abstrações identificadas como elementos de primeira classe em um modelo, os relacionamentos reúnem esses itens e os diagramas agrupam coleções interessantes de itens [BOO2000].

4.1 - Itens

Os itens constituem os blocos de construção básicos da UML e serão usados para escrever modelos bem-formados.

4.1.1 - Itens estruturais

São os itens estáticos do modelo, representando elementos conceituais ou físicos. Foram definidos sete elementos como pertencentes ao grupo de itens estruturais: *interface*, *classe*, *colaboração*, *caso de uso*, *classe ativa*, *componente* e *nó*.

Uma *interface* é um conjunto de operações que especificam serviços de uma classe ou componente, descrevendo o comportamento externamente visível desse

elemento. A interface define o conjunto de operações (assinaturas), mas não suas realizações.

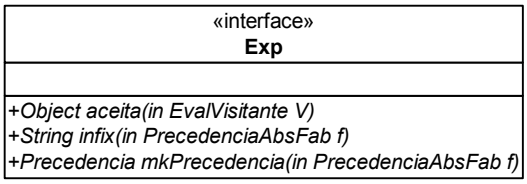


Figura 2.2 Representação gráfica de interface em UML.

A *classe* é a descrição de um conjunto de objetos que compartilham os mesmos atributos, operações, relacionamentos e semântica. Uma classe realiza uma ou mais interfaces. É representada graficamente como um retângulo com seu nome, atributos e operações conforme mostra a Fig. 2.3.

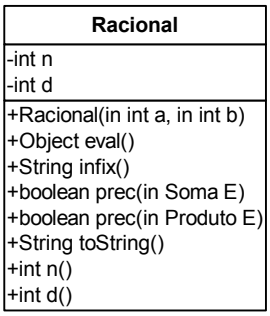


Figura 2.3 Representação gráfica de classe em UML.

As *colaborações* definem interações e são sociedades de papéis e outros elementos que funcionam em conjunto para proporcionar um comportamento cooperativo superior à soma de todos os elementos.

Um *caso de uso* é a descrição de um conjunto de seqüência de ações realizadas pelo sistema que proporciona resultados observáveis importantes para um determinado elemento participante do processo, normalmente chamado de ator.

As *classes ativas* são classes cujos objetos tem um ou mais processos ou *threads* e, portanto, podem iniciar a atividade de controle.

Os *componentes* são partes físicas e substituíveis de um sistema, que proporcionam a realização de um conjunto de interfaces. Normalmente os componentes representam o pacote físico de elementos lógicos diferentes, como classes, interfaces e colaborações. Graficamente um componente é representado por um retângulo com abas, incluindo somente seu nome, conforme mostra a Fig.2.4.

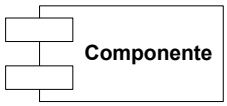


Figura 2.4 Representação gráfica de um componente em UML.

Um *nó* é um elemento físico existente em tempo de execução que representa um recurso computacional, geralmente com alguma memória e, freqüentemente, capacidade de processamento (cf. Fig 2.5). Um conjunto de componentes poderá estar contido em um nó e também poderá migrar de um nó para outro.

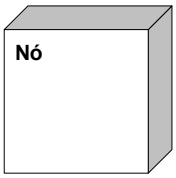


Figura 2.5 Representação gráfica de um nó em UML.

4.1.2 - Itens comportamentais

Representam as partes dinâmicas de um modelo em UML e seus comportamentos no tempo e no espaço. Geralmente são dois: *interação* e *máquina de estado*.

Uma *interação* é um comportamento que abrange um conjunto de mensagens trocadas entre um conjunto de objetos de um determinado contexto para a realização de propósitos específicos.

Uma *máquina de estado* é um comportamento que especifica as seqüências de estados pelas quais objetos ou interações passam durante sua existência em resposta a eventos, bem como suas respostas a esses eventos.

4.1.3 - Itens de agrupamento

São as partes organizacionais dos modelos de UML. São os blocos em que os modelos podem ser decompostos.

O *pacote* é o principal item de agrupamento e é um mecanismo de propósito geral para a organização de elementos em grupos (cf. Fig. 2.6). Ele é puramente conceitual e só existe em tempo de desenvolvimento. Uma variação do pacote é o framework, que será visto com mais detalhes no capítulo 3.

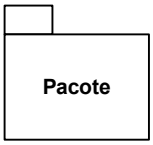


Figura 2.6 Representação gráfica de um pacote.

4.1.4 - Itens anotacionais

São as partes explicativas do modelo, sendo comentários para descrever, esclarecer ou fazer alguma observação sobre qualquer elemento. Seu representante é a *nota* (cf. Fig. 2.7), que funciona como um símbolo gráfico para agrupar os comentários.

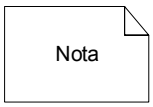


Figura 2.7 Representação gráfica do item anotacional.

4.2 - Relacionamentos

São os blocos relacionais básicos de construção da UML, e de suma importância para a construção de modelos bem-estruturados. Podem ser divididos em quatro tipos: *dependência*, *associação*, *generalização* e *realização*.

A *dependência* cria um relacionamento semântico entre dois itens, nos quais a alteração de um pode afetar a semântica do outro.

A *associação* é um relacionamento estrutural que descreve um conjunto de ligações, que são conexões entre objetos. A *agregação* é um tipo especial de associação, que representa um relacionamento estrutural entre um membro e seu grupo.

A *generalização* é um relacionamento de especialização/generalização, nos quais os objetos dos elementos generalizados são substituíveis por objetos do elemento especializado. Dessa maneira os objetos filhos (especializados) compartilham a estrutura e o comportamento dos pais (generalizados).

A *realização* é um relacionamento semântico entre classificadores, em que um classificador especifica um contrato que outro classificador garante executar. Normalmente são encontrados entre a interface e as classes ou componentes que a realizam e entre casos de uso e as colaborações que os realizam.

4.3 - Diagramas

Os *diagramas* são apresentações gráficas em forma de grafos que representam os elementos de um modelo sob uma determinada visão parcial. No total existem nove diagramas, mas a utilização de cada um depende da necessidade do modelo. Os tipos de diagramas são: *classes*, *objetos*, *casos de uso*, *seqüências*, *colaborações*, *estados*, *atividades*, *componentes* e *implantação*.

O *diagrama de classes* exibe um conjunto de classes, interfaces e colaborações, bem como seus relacionamentos. Um exemplo de diagrama de classes utilizado como exemplo no Capítulo 4 é apresentado na Fig. 2.8.

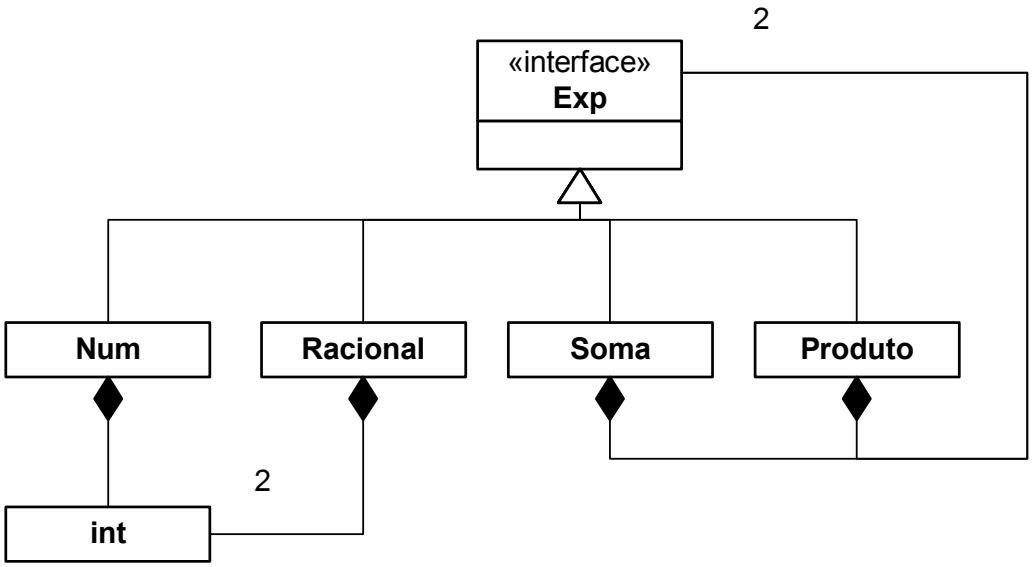


Figura 2.8 Diagrama de classes.

O *diagrama de objetos* exibe um conjunto de objetos e seus relacionamentos. São retratos estáticos de instâncias de itens encontrados em diagramas de classes.

O *diagrama de caso de uso* exibe um conjunto de casos de uso, atores e seus relacionamentos. São importantes para uma visão da organização e modelagem do comportamento do sistema.

O *diagrama de seqüências* é um diagrama de interação cuja ênfase está na ordenação temporal das mensagens e o *diagrama de colaboração* é um diagrama de interação cuja ênfase está na organização estrutural dos objetos que enviam e recebem mensagens.

O *diagrama de estados* exibe uma máquina de estados, formada por estados, transições, eventos e atividades e abrange uma visão dinâmica de um sistema.

O *diagrama de atividade* é um tipo especial de estado, exibindo o fluxo de uma atividade para outra no sistema.

O *diagrama de componente* exibe as organizações e as dependências existentes em um conjunto de componentes, abrangendo a visão estática da realização de um sistema. Tipicamente os componentes são mapeados para uma ou mais classes, interfaces ou colaborações.

O *diagrama de implantação* mostra a configuração dos nós de processamento em tempo de execução e os componentes neles existentes.

5 - Considerações sobre a modelagem usando objetos

Existem várias maneiras de se definir um modelo, mas no caso de um software, as duas abordagens mais comuns são: foco em algoritmos e foco em objetos. Historicamente, o foco em algoritmos foi muito utilizado, mas vem sendo gradativamente substituído pelo foco em objetos nos sistemas em todos os tipos de domínio de problemas, abrangendo todos os graus de tamanho e complexidade.

A utilização da abordagem orientada a objetos é inevitável para quem pensa em começar a desenvolver sistemas adequados ao nosso tempo. A adoção de uma linguagem padronizada para representar os resultados das análises dos modelos é imprescindível e a UML vem atender justamente essa necessidade.

Hoje existem ferramentas gráficas que facilitam bastante a vida do programador, mas não fazem o principal: montar a solução com a escolha correta das classes e suas estruturas de cooperação. O grande trabalho continua sendo o do mentor que está por trás de todas as soluções dos problemas deste mundo: o cérebro humano.

Capítulo 3

Padrões de Projeto

1 - Introdução

O processo de elaboração de software pode ter momentos em que a solução desejada já foi vista, analisada, projetada e desenvolvida por alguém em situação parecida. Isso nos leva a pensar se não existe uma maneira de compartilhar um conjunto de soluções de problemas previamente resolvidos, acessíveis e em um formato de fácil assimilação. Os *padrões de projeto* nasceram justamente para suprir esta necessidade.

O uso de padrões[1] pode fazer a diferença na qualidade do processo de desenvolvimento de software. O projetista de software pode e deve usar padrões para agilizar o tempo de desenvolvimento, sem a necessidade de redescobrir soluções já experimentadas com sucesso – o tal chamado "reinvento da roda". Maximizando as possibilidades de sucesso do produto final.

Este capítulo consacra-se ao estudo de padrões, conceitos e perspectivas de utilização.

2 - Padrões

"O conceito básico de um padrão também pode ser visto como o conceito básico de um sistema: adicionar uma camada de abstração entre o problema e a solução" [ECK2002]. Em [BOO2000] vemos uma outra definição de padrão: "Um padrão é uma solução comum para um problema básico em um determinado contexto". Christopher Alexander descreve padrões da seguinte maneira:

---

[1] Freqüentemente neste texto, utiliza-se a palavra "padrões" com o significado restrito a "padrões de projeto".

"Cada padrão descreve um problema no nosso ambiente e o núcleo da sua solução, de tal forma que você possa usar esta solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira" [GAM2000].

Partindo dessas idéias, podemos visualizar um padrão como um elemento auxiliar na realização de software que nos oferece uma interface entre um problema e uma solução anteriormente pensada, projetada, resolvida e testada.

Esses conceitos são verdadeiramente aplicados em muitas áreas de conhecimento. Em engenharia de software são importantes, pelos seguintes motivos:

- O uso de padrões facilita a reutilização de projetos e arquiteturas bem sucedidas. Expressar técnicas testadas e aprovadas as torna mais acessíveis para os projetistas de novos sistemas.
- Os padrões de projeto ajudam a escolher opções de projeto que tornam um sistema reutilizável e a evitar alternativas que comprometam a reutilização.
- Os padrões de projeto podem melhorar a documentação e a manutenção de sistemas ao fornecer uma especificação clara das interações entre classes e objetos e o seu objetivo subjacente.

Em suma, ajudam um projetista a obter um projeto correto e rápido [GAM2000].

Em todos os sistemas bem-estruturados, podemos encontrar uma grande variedade de padrões em diversos níveis de abstração. Padrões especificam a estrutura e o comportamento de uma sociedade de classes; os padrões de arquitetura especificam a estrutura e o comportamento de todo um sistema. Criando padrões explicitamente em um sistema, o tornamos mais compreensível e fácil de desenvolver e manter.

Os padrões nos ajudam a visualizar, especificar, construir e documentar os artefatos de um sistema complexo de software. Podemos usar a engenharia de produção para um sistema, selecionando um conjunto de padrões apropriados e aplicando-os às abstrações específicas do seu domínio. Ao finalizar um sistema, podemos especificar os padrões nele existentes para que, mais tarde, quando outra pessoa quiser reutilizá-lo ou adaptá-lo o faça com mais clareza e objetivo [BOO2000].

3 - Frameworks

Um *mecanismo* é um padrão de projeto aplicado a uma sociedade de classes e um *framework* é um padrão de arquitetura que fornece um modelo extensível a qualquer aplicação dentro de um domínio específico. Os padrões são utilizados para especificar os mecanismos e frameworks que compõem a arquitetura do sistema. [BOO2000].

Podemos pensar em um framework como um tipo de arquitetura básica abrangendo um conjunto de mecanismos que trabalham juntos para resolver uma classe específica de problemas. Ao especificar um framework, especificamos o esqueleto da sua arquitetura, juntamente com os conectores, guias, botões e indicadores que expomos aos usuários que desejam adaptá-lo ao seu próprio contexto.

Um framework captura as decisões de projeto que são comuns ao seu domínio de aplicação. Assim, frameworks enfatizam reutilização de projetos em relação à reutilização de código, embora um framework, geralmente, inclua subclasses concretas que podemos utilizar imediatamente.

Um framework montado através do uso de padrões tem muito maior possibilidade de atingir altos níveis de reusabilidade de projeto e código, comparado com um que não usa padrões. Frameworks maduros comumente incorporam vários padrões de projeto. Os padrões ajudam a tornar a arquitetura do framework adequada a muitas aplicações diferentes, sem necessidade de reformulação [GAM2000].

Será visto no Capítulo 4 um exemplo de framework para a avaliação de expressões matemáticas.

4 - Um exemplo: a arquitetura MVC

A arquitetura MVC (Modelo / Visual / Controlador – “Model / View / Controler”) tem sua origem no SmallTalk, onde foi aplicada para mapear as tradicionais tarefas de entrada, processamento e saída para interfaces gráficas do usuário. Em seguida, a arquitetura MVC foi adaptada para mapear os mesmos conceitos em domínio de aplicações multi-camadas.

A arquitetura MVC, de fato um padrão, é composto por três tipos de objetos. O modelo é o objeto de aplicação, o visual é a apresentação na tela e o controlador define a maneira pela qual a interface gráfica reage às entradas do usuário. Antes do padrão MVC, os projetos de interface gráfica tendiam a agrupar esses objetos. O padrão MVC os separa para aumentar a flexibilidade e a reutilização [GAM2000].

O *modelo* representa os dados da aplicação e as regras de negócios que norteiam o acesso e atualização dos dados. Geralmente o modelo serve como uma aproximação do software para o processo do mundo real, logo, quando definimos o modelo, aplicamos as técnicas de modelagem do mundo real.

O *visual* é o responsável por mostrar o conteúdo do modelo. Acessa os dados da aplicação através do modelo e especifica como esses dados devem ser apresentados. É sua responsabilidade manter a consistência da apresentação quando existe alguma mudança no modelo.

O *controlador* traduz as interações do visual para ações a serem executadas pelo modelo. As ações exercidas pelo modelo incluem ativações de processos de negócios ou mudanças no estado do modelo. Baseado nas interações do usuário e nos resultados das ações do modelo, o controlador responde selecionando o visual apropriado.

A arquitetura MVC traz os seguintes benefícios [SUN2002]:

Múltiplas visões usando um mesmo modelo:

A separação entre modelo e visual permite que múltiplos visuais possam ser usados em um mesmo modelo. Conseqüentemente, o modelo de componentes de um aplicação fica mais fácil de realizar, testar e manter, desde que todos os acessos ao modelo sejam efetuados pelos componentes.

Suporte facilitado para novos tipos de clientes:

Para dar suporte a mais um cliente, basta desenvolver um visual e um controlador e encaixá-los no modelo existente.

Um exemplo de aplicação da arquitetura MVC é visto em [GAM2000]. Um modelo contendo alguns valores de dados e três visuais, contendo uma planilha, um histograma e um gráfico de pizza, apresentam os dados de várias maneiras conforme Fig. 3.1.

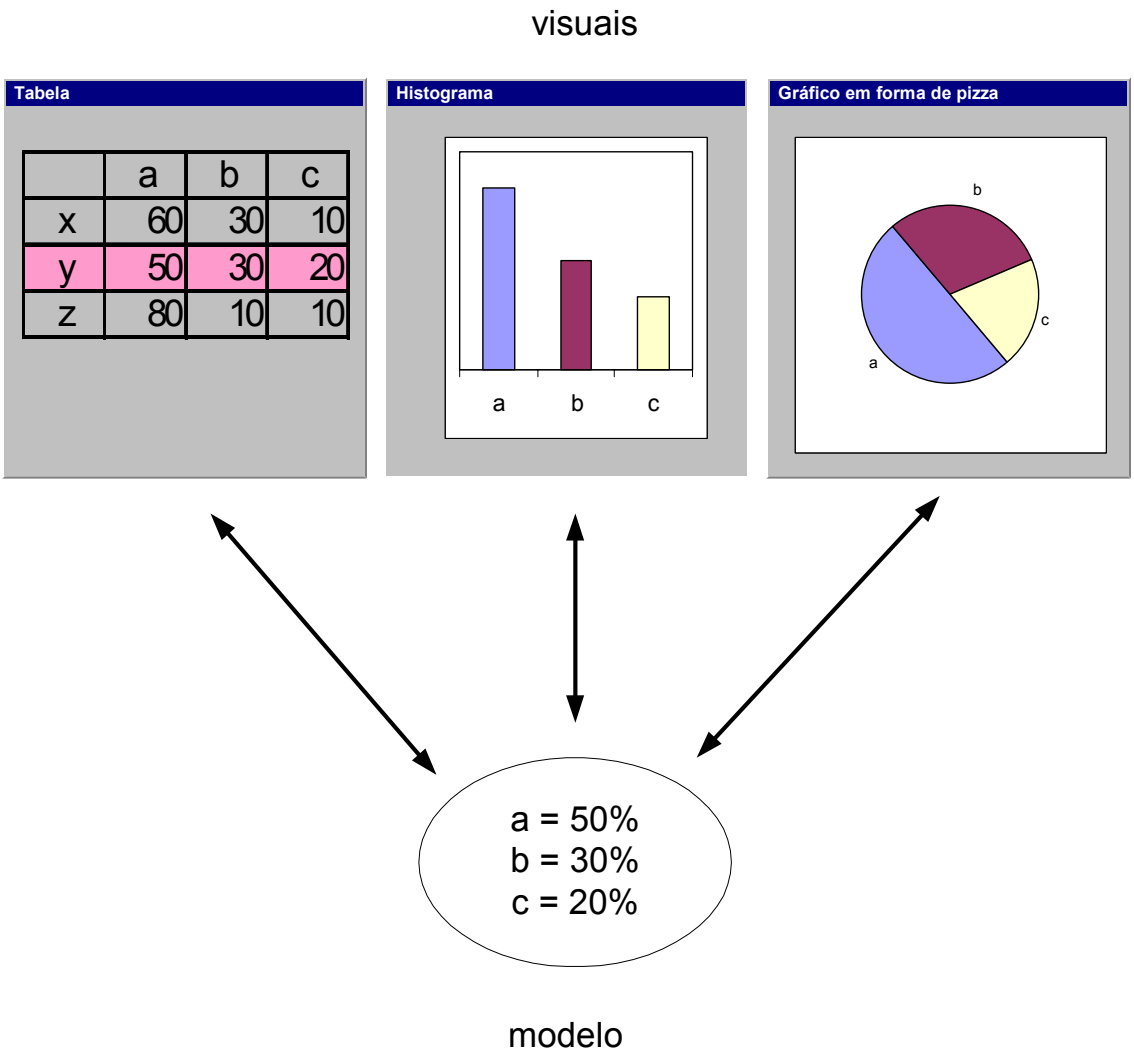


Figura 3.1 Exemplo de aplicação da arquitetura MVC [GAM2000].

Um modelo relaciona-se com seus visuais quando seus valores se modificam, e os visuais se comunicam com o modelo para acessar os valores que mostrarão, os controladores foram omitidos por simplificação.

5 - Descrevendo padrões de projeto

Normalmente um padrão de projeto possui quatro elementos essenciais [GAM2000]:

1. Nome do padrão

É uma referência que podemos usar para descrever um problema de projeto, suas soluções e conseqüências em uma ou duas palavras. Dar nome a um padrão aumenta imediatamente o nosso vocabulário de projeto. Isso nos permite projetar em um nível mais alto de abstração. Ter um vocabulário para padrões permite-nos conversar sobre eles com nossos colegas e referenciá-los em nossa documentação. O nome torna mais fácil o pensar sobre projetos e a comunicá-los, bem como os custos e benefícios envolvidos, a outras pessoas.

2. Problema

Descreve quando aplicar o padrão. Ele explica o problema e o seu contexto. Pode descrever problemas de projeto específicos, tais como representar algoritmos como objetos. Pode descrever a estrutura de classes ou objetos sintomáticos de um projeto. Algumas vezes, o problema incluirá uma lista de condições que devem ser satisfeitas para que faça sentido aplicar o padrão.

3. Solução

Descreve os elementos que compõem o projeto, seus relacionamentos, suas responsabilidades e colaborações. A solução não descreve um projeto concreto ou uma realização em particular porque um padrão é como um gabarito que pode ser aplicado em muitas situações diferentes. Em vez disso, o padrão fornece uma descrição abstrata de um problema de projeto e de como um arranjo geral de elementos (classes e objetos, no nosso caso) resolve o mesmo.

4. Conseqüências

Fornece o resultado e análise das vantagens e desvantagens da aplicação do padrão. Embora as conseqüências sejam raramente mencionadas quando descrevemos decisões de projeto, elas são críticas para a avaliação de alternativas de projetos e para a compreensão dos custos e benefícios da aplicação do padrão. As conseqüências para o

software freqüentemente envolvem compromissos de espaço e tempo. Elas também podem abordar aspectos sobre linguagens e realização. Uma vez que a reutilização é freqüentemente um fator no projeto orientado a objetos, as conseqüências de um padrão incluem o seu impacto sobre a flexibilidade, a extensibilidade ou a portabilidade de um sistema. Relacionar estas conseqüências explicitamente ajuda a compreendê-las e avaliá-las.

A descrição de um padrão de projeto envolve os elementos que o compõem, suas funções e documentações. Além dos quatro elementos essenciais vistos a pouco, podemos também utilizar outras formas de descrição de um padrão. A forma apresentada a seguir é conhecida pelo nome de "formato GoF" [GAM2000]:

Nome e classificação do padrão

O nome do padrão expressa a sua própria essência de forma sucinta. Um bom nome é vital, porque ele se tornará parte do seu vocabulário de projeto. A classificação do padrão descreve a que categoria, por escopo ou finalidade, ele pertence.

Intenção e objetivo

É uma curta declaração que explica o que faz o padrão de projeto, quais os seus princípios e sua intenção e que tópico ou problema particular de projeto ele trata e se propõe a ajudar resolver.

Também conhecido como

Outros nomes bem conhecidos para o padrão, caso existam.

Motivação

Um cenário que ilustra um problema de projeto e como as estruturas de classes e objetos no padrão solucionam o problema.

Aplicabilidade

Mostra quais as situações nas quais o padrão de projeto pode ser aplicado, que exemplos de maus projetos ele pode tratar e como podemos reconhecer essas situações.

Estrutura

Uma representação gráfica das classes do padrão.

Participantes

As classes e/ou objetos que participam do padrão de projeto e suas responsabilidades.

Colaborações

Como os participantes colaboram para executar suas responsabilidades.

Conseqüências

Mostra como o padrão suporta a realização de seus objetivos, quais são os seus custos e benefícios, os resultados da sua utilização e que aspecto da estrutura do sistema ele permite variações.

Realização

Mostra que armadilhas, sugestões ou técnicas você precisa conhecer quando da realização do padrão e se existem considerações específicas de linguagem.

Exemplo de código

Fragmentos ou blocos de código que ilustram como podemos programar o padrão em uma linguagem de programação adequada, por exemplo, Java, C++ ou SmallTalk.

Usos conhecidos

Exemplos do padrão encontrados em sistemas conhecidos.

Padrões relacionados

Mostra quais padrões de projeto estão intimamente relacionados com este, quais são as diferenças importantes e com quais outros padrões este deveria ser usado.

O formato GoF[2] é o mais utilizado e contempla os elementos essenciais de um padrão. Pode-se, é claro, se necessário, envolver outras características de um determinado padrão quando da sua descrição ou divulgação.

6 - O catálogo GoF

Os principais padrões para projeto de software foram catalogados em [GAM2000]. Contém a descrição de vinte e três padrões, são eles:

Adaptador ("Adapter")

Converte a interface de uma classe em outra interface mais adequada às classes clientes. O padrão adaptador permite que certas classes trabalhem em conjunto, pois de outra forma seria impossível por causa de suas interfaces incompatíveis. Isso permite que classes possuindo interfaces incompatíveis entre si adaptem-se umas as outras para um trabalho cooperativo.

Aproximador ("Proxy")

Fornece um objeto representante, ou um marcador de outro objeto, para controlar o acesso ao mesmo.

Cadeia de Responsabilidade ("Chain of Responsibility")

Evita o acoplamento do remetente de uma solicitação ao seu destinatário, dando a mais de um objeto a chance de tratar uma determinada solicitação. Encadeia os objetos receptores e passa a solicitação ao longo da cadeia até que um objeto possa tratá-la.

Comando ("Command")

Encapsula uma solicitação como um objeto, desta forma permitindo a parametrização de clientes com diferentes solicitações, enfileire ou registre solicitações e suporte operações que possam ser desfeitas.

---

[2] GoF vem do inglês Gang of Four e é usado como referência ao grupo composto por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides.

Composto (“Composite”)

Compõe objetos em estrutura de árvore para representar hierarquias do tipo parte-todo. O padrão composto permite que os clientes tratem objetos individuais e composições de objetos de maneira uniforme.

Conjunto Unitário (“Singleton”)

Garante que uma classe tenha somente uma instância e fornece uma referência global de acesso.

Construtor (“Builder”)

Separa a construção de um objeto complexo da sua representação, de modo que o mesmo processo de construção possa criar diferentes representações.

Decorador (“Decorator”):

Atribui responsabilidades adicionais a um objeto dinamicamente. O padrão Decorador fornece uma alternativa flexível a subclasses para extensão da funcionalidade.

Estado (“State”)

Permite que um objeto altere seu comportamento quando seu estado interno é alterado. O objeto parecerá ter alterado sua classe.

Estratégia (“Strategy”)

Define uma família de algoritmos, encapsula cada um deles e os faz intercambiáveis. O padrão Estratégia permite que o algoritmo varie independentemente dos clientes que o utilizam.

Fábrica Abstrata (“Abstract Factory”)

Fornece uma interface para a criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.

Fachada (“Façade”)

Fornece uma interface unificada para um conjunto de interfaces em um subsistema. O padrão Fachada define uma interface de nível mais alto que torna o subsistema mais fácil de usar.

Interpretador (“Interpreter”)

Dada uma linguagem, define uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças nesta linguagem.

Iterador (“Iterator”)

Fornece uma maneira de acessar seqüencialmente os elementos de um objeto agregado sem expor sua representação interna.

Lembrete (“Memento”)

Sem violar a encapsulação, captura e externaliza um estado interno de um objeto, de modo que o mesmo possa posteriormente ser restaurado.

Mediador (“Mediator”)

Define um objeto que encapsula como um conjunto de objetos faz suas interações. O padrão mediador promove o acoplamento fraco entre classes ao evitar que os objetos refiram-se explicitamente uns aos outros, permitindo que suas interações variem independentemente.

Método de Fábrica (“Factory Method”)

Define uma interface para criar um objeto, mas deixa a cargo das subclasses a decisão da instanciação. O padrão método de fábrica permite a uma classe postergar a instanciação às subclasses.

Método de Molde (“Template Method”)

Define o esqueleto de um algoritmo associado a uma operação, postergando a definição de alguns passos para subclasses. Permite que as subclasses redefinam certos passos de um algoritmo sem modificar, no entanto, sua estrutura.

Observador (“Observer”)

Define uma dependência um-para-muitos entre objetos, de modo que, quando um objeto modifica o seu estado, todos os seus dependentes são automaticamente notificados e atualizados.

Peso Leve (“Flyweight”)

Usa compartilhamento para suportar grandes quantidades de objetos, de granularidade fina, de maneira eficiente.

Ponte (“Bridge”)

Separa uma abstração da sua realização, de modo que as duas possam variar independentemente.

Protótipo (“Prototype”)

Especifica os tipos de objetos a serem criados usando uma instância particular como protótipo. A criação de novos objetos dar-se-á pela cópia do protótipo.

Visitante (“Visitor”)

Representa uma operação a ser executada sobre os elementos da estrutura de um objeto. O padrão visitante permite que possamos definir uma nova operação sem modificar as classes dos elementos sobre os quais opera.

Todos esses padrões podem ser agrupados por famílias de padrões relacionados, de acordo com o seu *propósito* e *escopo*.

Na visão de seu *propósito*, a finalidade é levada em consideração: *criação*, *estrutural* e *comportamental*. Um padrão classificado como de *criação* preocupa-se com o processo de criação dos objetos. Um padrão *estrutural* lida com a composição de classes ou de objetos. Um padrão comportamental caracteriza a maneira pela qual classes ou objetos interagem e distribuem responsabilidades.

Na visão de seu *escopo*, é verificado se o padrão se aplica primariamente a *classes* ou *objetos*. Os padrões para *classes* lidam com relacionamentos entre classes e suas subclasses. Esses relacionamentos são estabelecidos através do mecanismo de herança, assim eles são estáticos e fixados em tempo de compilação. Os padrões para *objetos* lidam com relacionamentos entre objetos que podem ser modificados em tempo de execução e são dinâmicos. Quase todos utilizam a herança em certa medida. A maioria dos padrões está no escopo de Objeto.

A Tab. 3.1 mostra a classificação dos padrões em relação ao propósito e ao escopo.

		Propósito		
		Criação	Estrutural	Comportamental
Escopo	Classe	Método de Fábrica	Adaptador	Interpretador Método de Molde
	Objeto	Conjunto Unitário Construtor Fábrica Abstrata Protótipo	Adaptador Aproximador Composto Decorador Fachada Peso Leve Ponte	Cadeia de Responsabilidade Comando Estado Estratégia Iterador Lembrete Mediador Observador Visitante

Tabela 3.1 Classificação dos padrões GoF [GAM2000].

7 - Utilizando padrões

7.1 - O padrão Estado

O padrão Estado é classificado como comportamental de objeto e permite a um objeto alterar seu comportamento quando o seu estado interno se modifica, fazendo com que pareça que o objeto modificou sua própria classe [GAM2000].

Pode ser usado quando o comportamento do objeto modifica-se em tempo de execução ou em situações em que o comportamento de um objeto tenha várias alternativas ou obedeça a várias condições. Nesse caso, o padrão Estado coloca cada trecho do comando adicional em uma classe separada. Em ambas as situações os estados dos objetos são tratados como objetos propriamente ditos.

Seus participantes são:

Contexto: define a interface de interesse para os clientes e mantém uma instância de uma subclasse *EstadoConcreto* que define o estado atual.

Estado: define uma interface para encapsulamento associado a um determinado estado do *Contexto*.

Subclasse EstadoConcreto: cada subclasse realiza um comportamento associado a um estado do *Contexto*.

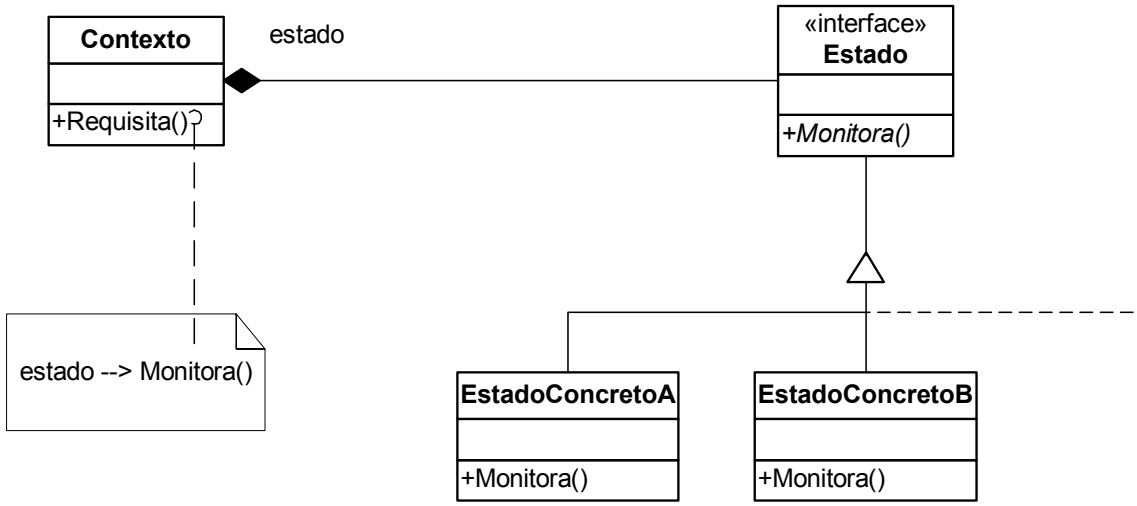


Figura 3.2 Visão geral do padrão Estado e seus elementos

A Fig. 3.2 ilustra os elementos acima citados e seus relacionamentos.

O *Contexto* delega solicitações específicas de estados para o objeto atual *EstadoConcreto*. Um contexto pode passar a si próprio como argumento para o objeto *Estado* que trata a solicitação. Isso permite ao objeto *Estado* acessar o *Contexto*, se necessário.

O *Contexto* define uma interface primária para os clientes. Os clientes podem configurar um contexto com objetos *Estado*. Uma vez que o contexto está configurado, seus clientes não têm que lidar com os objetos *Estado* diretamente.

Tanto o *Contexto* quanto as subclasses de *EstadoConcreto* podem decidir a mudança de estado e sob quais circunstâncias.

O uso do padrão estado pode gerar as seguintes conseqüências [GAM2000]:

1) Confinar o comportamento específico de estados e particionar o comportamento para estados diferentes. O padrão coloca todo o comportamento

associado a um estado particular em um objeto. Novos estados e transições de estado podem facilmente ser adicionados pela definição de novas subclasses.

2) Tornar explícitas as transições de estado.

3) Objetos *Estado* podem ser compartilhados.

O padrão Estado focaliza sobre como lidar com um objeto cujo comportamento depende de seu estado. Seu uso facilita a realização de problemas dependentes de estado, com ênfase nos seus estados e suas transições de estado.

7.2 - Exemplo: a calculadora

Para ilustrar a aplicação dos benefícios do uso dos padrões, mostra-se como a utilização do padrão Estado pode significativamente melhorar o projeto de uma calculadora contemplando as operações de soma, subtração, multiplicação e divisão (cf. Fig. 3.3).

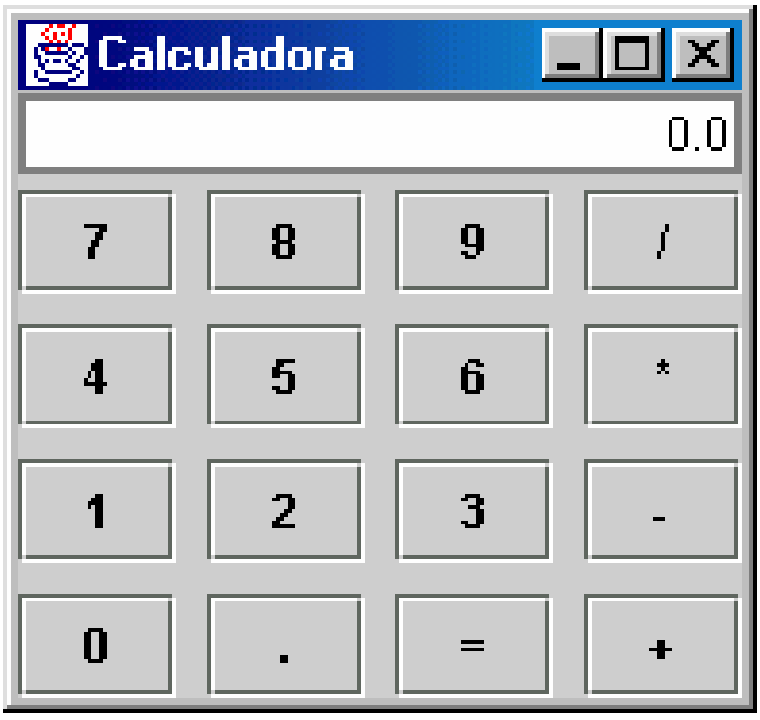


Figura 3.3 Janela do aplicativo calculadora.

O diagrama de estados apresentado na Fig. 3.4 explicita as três principais atividades da calculadora e suas transições.

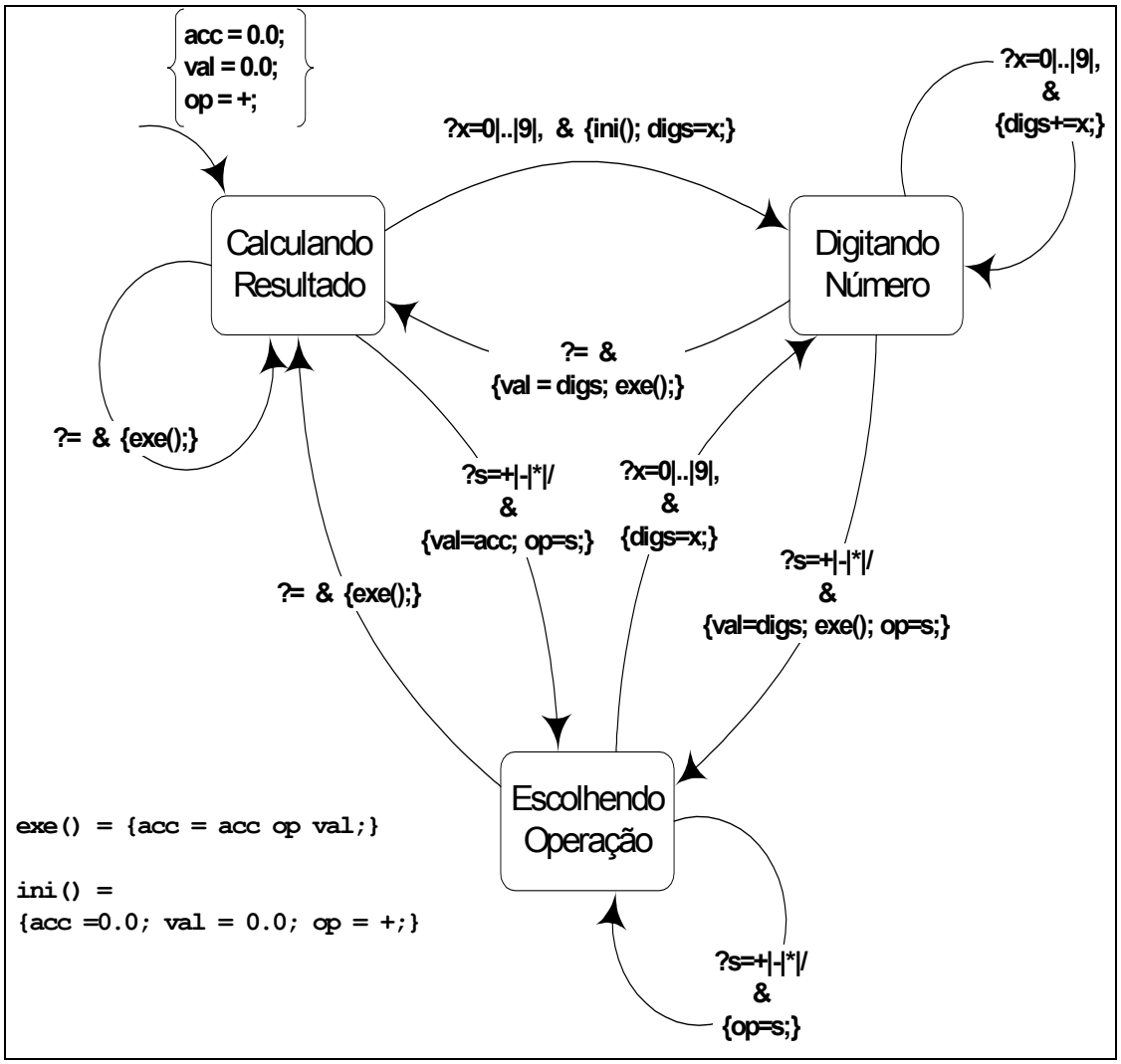


Figura 3.4 Diagrama de estados do aplicativo calculadora.

Resumidamente a calculadora pode estar recebendo, através de botões, comandos para representar dígitos numéricos ([0..9]), o ponto (“.”) para representar a parte decimal de um número, o símbolo de igualdade (“=”) ou operadores algébricos (“+”, “-“, “*”, “/”). Os principais valores controlados pelo programa são: acc (acumulador), op (operador) e val (último valor digitado).

A calculadora pode assumir três estados: *digitando número*, *escolhendo operação* ou *calculando resultado*. Quando a calculadora estiver no estado *digitando número*, ela deve acumular os valores até receber um dígito de operação, causando uma mudança do seu estado para *escolhendo operação*. Se novamente a entrada for de dígitos numéricos, ela volta ao estado *digitando número* e espera a próxima entrada. No

momento em que for pressionado o botão de igualdade, ela passa para o estado *calculando resultado* e mostra o resultado obtido da operação escolhida.

Podemos perceber que a partir de cada estado, a calculadora pode permanecer no mesmo ou mudar para qualquer um dos outros.

Serão apresentadas duas versões da calculadora: com e sem o uso do padrão Estado.

Para todas as duas versões, foi utilizada uma mesma interface com o usuário chamada GUI (Fig. 3.5) que fornece os componentes e botões para representar graficamente a calculadora, receber as entradas e mostrar as saídas.

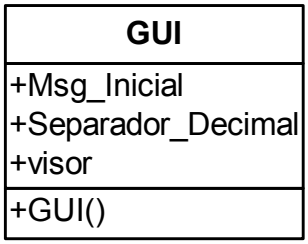


Figura 3.5 Representação da classe GUI.

7.3 - Solução sem o padrão *Estado*

Em uma primeira versão será mostrada uma solução sem a aplicação do padrão Estado, baseado em uma arquitetura (cf. Fig. 3.6) onde a mudança de estado é controlada dentro da classe principal.

No momento em que é pressionado o botão de igualdade, o programa executa a operação `acc = acc op val`, achando o novo valor de `acc` e mostrando no visor da calculadora o resultado obtido.

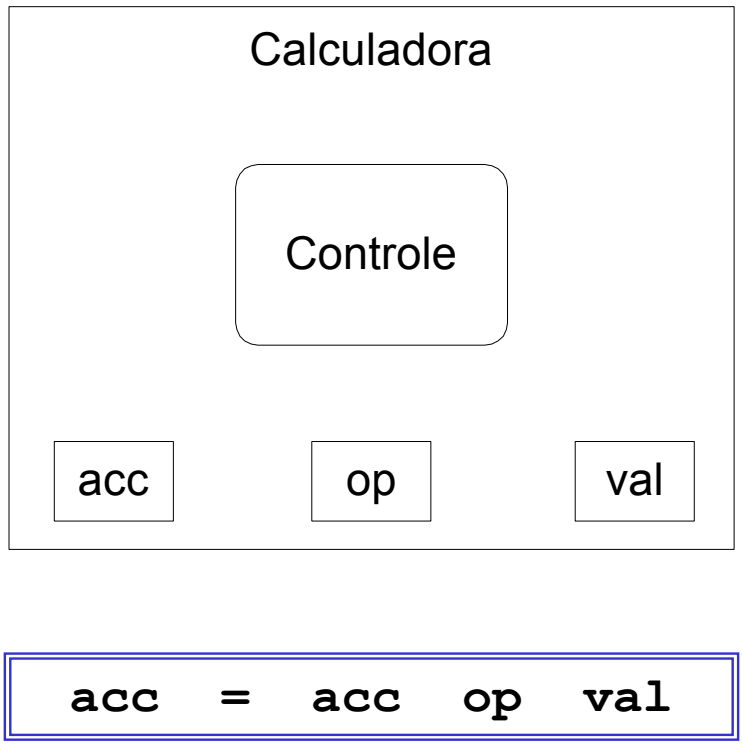


Figura 3.6 Estrutura da primeira solução.

A representação da classe calculadora (Fig. 3.7) mostra que as mudanças de estado são controladas internamente e são representadas por um atributo da classe que tem seu valor modificado quando há uma mudança no estado da calculadora.

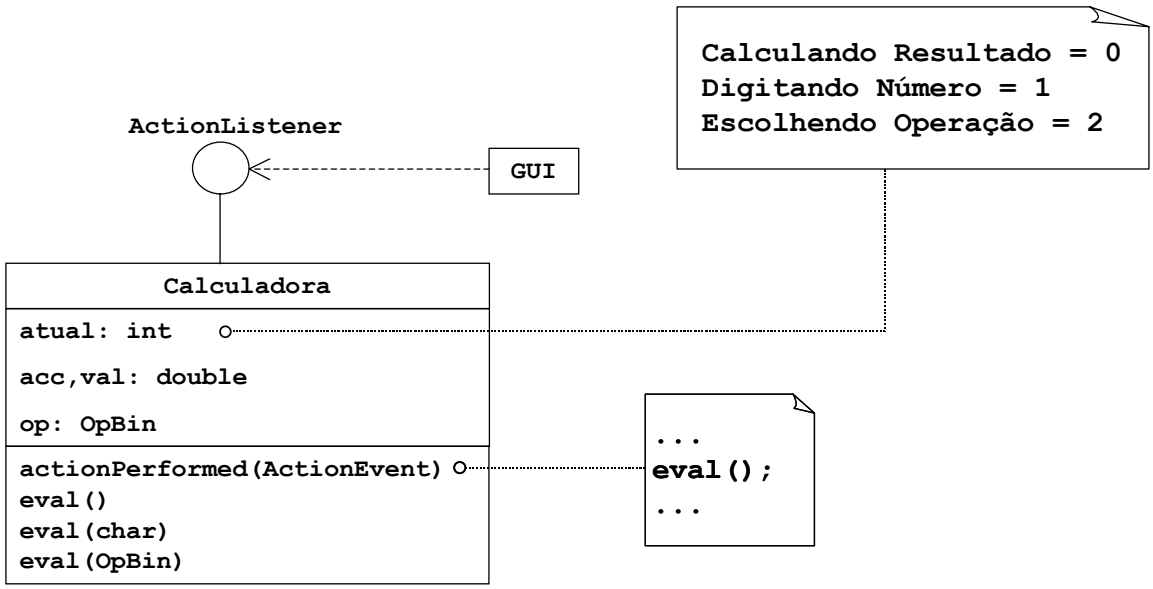


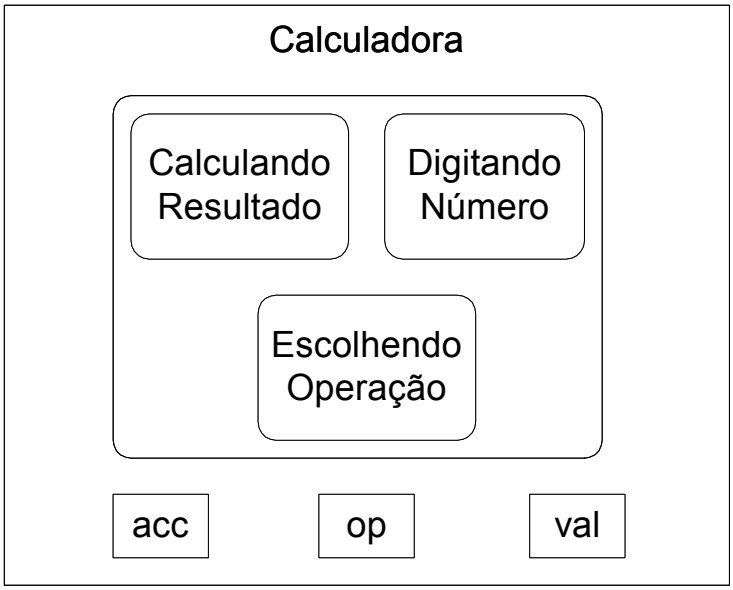
Figura 3.7 Representação da classe na primeira solução.

O grande inconveniente desta representação é que o comportamento da calculadora é descrito pelos métodos `eval()`, `eval(char)` e `eval(OpBin)` não importando o estado em que a calculadora encontra-se. Portanto, há a necessidade de refazer o código no caso de alguma alteração no projeto.

7.4 - Solução com o padrão *Estado*

O padrão Estado permite uma visão arquitetural da calculadora significativamente melhor que a solução anterior. As atividades não estarão mais dissimuladas no código, mas assumindo um papel explícito na estrutura comportamental da calculadora da forma como é ilustrada na Fig. 3.8.

O padrão Estado fornece uma interface chamada de estado que tem a responsabilidade de instanciar os três estados possíveis para a calculadora (cf. Fig. 3.9).




```
acc = acc op val
```

Figura 3.8 Estrutura da segunda solução.

Nessa nova arquitetura o objeto da classe Calculadora passa a alterar seu comportamento quando o seu estado interno se modifica. Suas ações dependem agora do estado atual. Pode-se notar que com este novo projeto, o comportamento da cada estado de funcionamento da calculadora, fornecido pelos objetos Estado, é dissociado do comportamento da classe Calculadora.

Note que como os estados são classes que realizam a interface Estado, suas instâncias são objetos separados da instância da classe Calculadora.

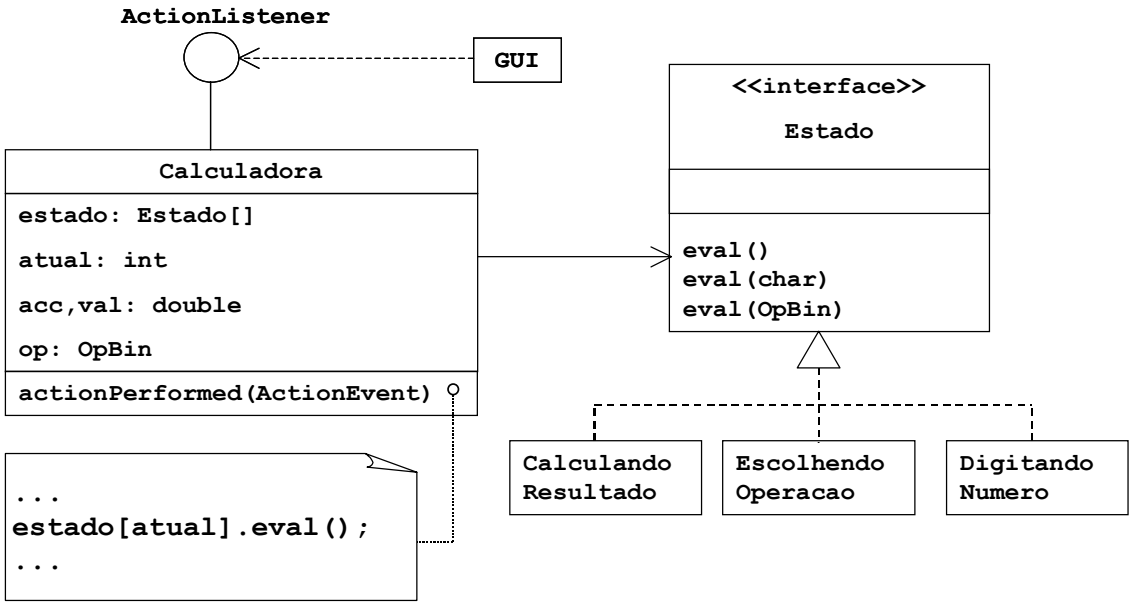


Figura 3.9 Representação da classe Calculadora e interface Estado.

A modificação do projeto para que a calculadora exiba uma nova funcionalidade, por exemplo, memorização, pode ser facilmente estabelecida pelo acréscimo de um novo tipo de objeto Estado, sem precisar modificar o comportamento associado aos outros tipos de objetos Estado. Isso melhora o grau de reusabilidade do projeto, tornando-o mais flexível e adaptável a novas funcionalidades.

8 - Conclusão

Existem padrões de fácil identificação e visualização que podem ter imediata aplicação e existem padrões que exigem uma maior experiência do projetista.

Um grande número de padrões está descrito e disponível para uso de acordo com a necessidade, a partir de um modelo de problema a ser resolvido e de sua solução.

Como um exemplo de aplicação de padrão, este capítulo apresentou o projeto de uma calculadora simples. Vimos como o padrão Estado foi positivamente aplicado a este caso, capturando o essencial da sua estrutura comportamental.

No próximo capítulo, aprofundaremos ainda mais este estudo sobre padrões apresentando um projeto de um sistema que incorpora na sua íntegra o principal intuito da aplicação de padrões que é a reusabilidade.

Capítulo 4

Aplicando Padrões de Projeto no Desenvolvimento de Frameworks

1 - Introdução

Este capítulo aborda o tema principal dessa dissertação que é a utilização de padrões no desenvolvimento de software orientado a objetos. É apresentado um estudo de caso voltado a construção e a avaliação de expressões matemáticas. A aplicação em si não é muito importante neste estudo, mas a utilização de padrões na solução de problemas de projeto relacionado diretamente à aplicação que, por sua vez, reflete uma problemática bastante recorrente em software.

Muito do que se conhece sobre a abordagem orientada a componentes no desenvolvimento de software refere-se geralmente a modelos arquiteturais de software já bem definidos. É caso, por exemplo, de software para o desenvolvimento de aplicações web (servlets e EJB da Sun Microsystems, DCOM da Microsoft, etc.) que mascaram grande parte das especificidades de suas arquiteturas. O foco do estudo apresentado neste capítulo é a utilização de padrões no desenvolvimento de um modelo de utilização de componentes ao software. Por isso, nem todos os conceitos associados a componentes serão verdadeiramente considerados.

O estudo parte com a definição da problemática na Seção 2; propõe uma primeira solução utilizando o padrão *Composto* na Seção 3; na Seção 4 é apresentado uma segunda solução com uso do padrão *Visitante*; a terceira versão da solução, na Seção 5, apresenta o padrão *Método de Fábrica* e finalmente a quarta versão, Seção 6, propõe o uso do padrão *Fábrica Abstrata*. A conclusão para este capítulo encontra-se na seção 7.

2 - A problemática

A avaliação de expressões matemáticas exige primeiramente uma representação adequada das expressões em termos de objetos de software. Para entendermos este problema, consideraremos inicialmente um conjunto simples de expressões compreendendo duas operações, adição e multiplicação, e apenas dois tipos de números, inteiros e racionais. Mais adiante discutiremos como estender este conjunto com um número maior de operações e números.

Expressões desse tipo podem ser concretamente escritas na tela de um computador da seguinte maneira:

5	(E1)
2/3	(E2)
5 * 3 + 4	(E3)
2 + 4/3	(E4)

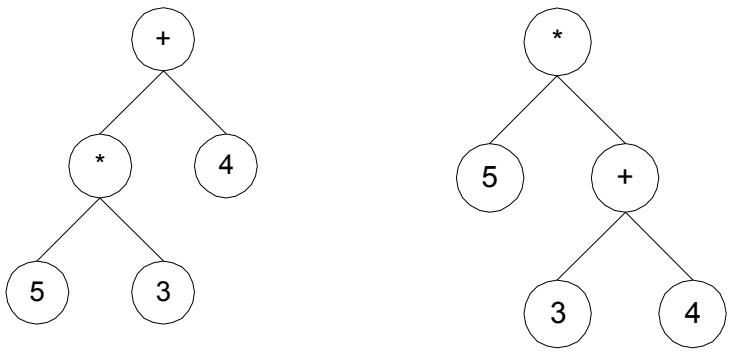
A expressões E1 e E2 representam apenas valores, embora E2 seja na realidade uma composição de dois valores inteiros. Na avaliação de uma expressão matemática, valores representam *expressões terminais*, possuem significado próprio, mas que podem, entretanto ser representado de mais de uma maneira – 2/3, 4/6, 8/12, ..., representam o mesmo número racional. Números racionais com numerador maior que o denominador são habitualmente representados na forma de número misto k n/d, onde k n/d = k + n/d = (k*d + n)/d . Por exemplo, 5/3 pode ser representado como 1 2/3, pois 1*3+2 = 5. Para o nosso caso, a forma de representação não é de fato importante. Entretanto, devemos ter em mente que a proposta que seguiremos neste capítulo, o racional 1/3 é diferente de 0,333 ... 33. Não importa o grau de precisão que estejamos trabalhando.

A expressões E3 e E4 por outro lado são composições de operações sobre valores e outras expressões e, por isso, precisam ser avaliadas. Representam *expressões não terminais*. A avaliação de uma expressão não necessariamente deve fornecer o mesmo tipo de valor. Por exemplo, a avaliação da expressão E4 = 2 + 4/3 pode resultar em um inteiro 3, avaliação inteira, em um racional 10/3, avaliação racional, ou em um

decimal 3,33, avaliação decimal. O primeiro e o terceiro resultado são aproximados, pois o valor correto da expressão é o racional 10/3.

Um outro fato importante sobre a representação de expressões são as simplificações sintáticas a partir da aplicação de prioridades diferentes às operações. Por exemplo, a expressão E3 é de fato uma representação ambígua, pois podemos ter duas interpretações: (5 * 3) + 4 ou 5 * (3 + 4). É considerando que a multiplicação tem prioridade sobre a soma que podemos resolver essa ambigüidade: 5 * 3 + 4 = (5 * 3) + 4 = 19.

Um dos principais problemas a ser resolvido pelo projeto é a representação de expressões em objetos de software. Uma maneira de fazer isso é converter as expressões na sua forma concreta – sintaxe concreta – na forma de uma árvore – sintaxe abstrata. A representação em árvore de uma expressão considera dois tipos de elementos: nós e folhas. Nós são elementos compostos de uma ou mais expressões, as ramificações do nó. É o caso, por exemplo, da adição e multiplicação. As folhas são simplesmente nós sem ramificações, portanto, representam valores. Assim, a expressão E3 acima pode ser abstratamente representada como mostra a Fig. 4.1.a. Podemos notar que a expressão 5 * (3 + 4) possui como esperado uma representação em árvore, Fig. 4.1.b, diferente da expressão E3.



(a) 5 * 3 + 4 (b) 5 * (3 + 4)

Figura 4.1 Sintaxe abstrata.

O processo da avaliação de expressões opera exclusivamente com expressões representadas na sua forma abstrata. Entretanto, há casos onde o resultado da avaliação de uma expressão é uma outra expressão. Por exemplo, a avaliação da expressão

d (x^2 + 3 * x + 5) / dx

descrevendo a derivada de x^2 + 3 * x + 5 com relação a x, tem como resultado a expressão 2*x + 3. Para externar adequadamente o resultado para o usuário, a problemática deve também envolver o problema da representação concreta de expressões.

O projeto do software que será desenvolvido deverá prover soluções para os seguintes problemas relacionado à representação e avaliação de expressões:

1. Editar expressões;
2. Representar expressões em sintaxe abstrata – objetos de software;
3. Fornecer diferentes formas de avaliação de expressões;
4. Representar expressões em sintaxe concreta.
5. Prover um ambiente de interação com o usuário.

O projeto apresentado neste capítulo fornece soluções apenas para os problemas 2, 3 e 4. A edição de expressões e a provisão de um ambiente usuário, considerados problemas ortogonais aos outros três, são colocados aqui como propostas de continuação do trabalho de dissertação.

O projeto apresentado na próxima seção contempla principalmente o problema da representação abstrata das expressões e propõe uma arquitetura centrada no padrão Composto. O problema 3 é propriamente tratado no projeto apresentado na Seção 4 utilizando o padrão Visitante. A solução adotada fornece uma estrutura orientada a componentes para a avaliação de expressões. O problema da representação de expressões em sintaxe concreta é abordado na Seção 6 utilizando o padrão *Fábrica Abstrata*.

3 - Solucionando o problema da representação de expressões – versão 0.1

Esta seção apresenta uma primeira versão do projeto do avaliador que basicamente fornece uma solução ao problema da representação de expressões utilizando o padrão Composto já mencionado no Capítulo 3. Para os outros problemas, esta versão oferece apenas soluções temporárias.

O padrão Composto é colocado como uma solução geral aos problemas de composição de objetos – hierarquias parte/todo – que podem ser estruturados na forma de uma árvore. Ele permite que clientes possam tratar coleções de objetos como objetos individuais [GAM2000]. Esse tipo de estrutura de composição é muito comum em software. Um exemplo típico é o modelo arquivo-diretório utilizado pelos sistemas operacionais (cf. Fig. 4.2).

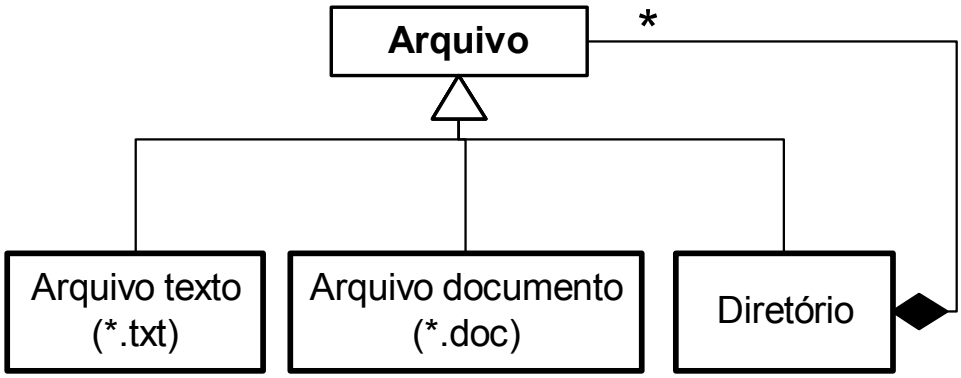


Figura 4.2 O modelo arquivo-diretório

Neste modelo, a estrutura mais simples é o arquivo genérico, que pode ter novas reclassificações, a medida em que precisamos de arquivos especializados. Um arquivo pode ser do tipo texto simples, pode ser um documento ou pode ainda ser um tipo especial de arquivo, chamado diretório, que por sua vez é uma composição de arquivos.

Esse exemplo caracteriza a natureza recursiva da estrutura de árvore. E é justamente essa característica que o padrão Composto explora.

Essa é a estrutura de composição exibida pelas expressões. Uma expressão compõe-se de outras expressões que, por sua vez, também compõem-se de outras expressões e assim por diante. A Fig. 4.3 apresenta um esboço dessa arquitetura de composição aplicada ao caso das expressões.

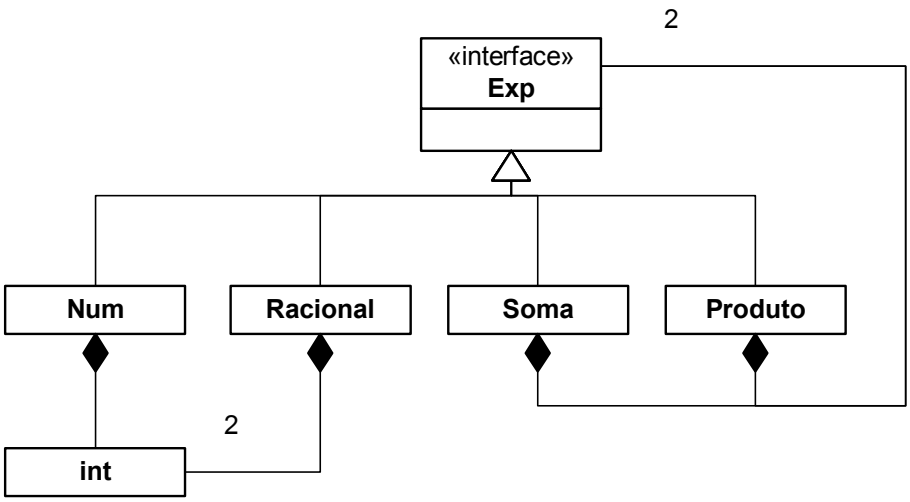


Figura 4.3 Modelo de composição de expressões

A interface **`Exp`** é utilizada para generalizar a composição recursiva dos elementos envolvidos na expressão. Assim, números inteiros (**`Num`**), números racionais (**`Racional`**), adições (**`Soma`**) ou multiplicação (**`Produto`**) são modeladas como simples realizações da interface **`Exp`**. Notemos que objetos Soma e Produto possuem dois atributos para objetos **`Exp`** quaisquer. Por exemplo, a expressão 5 * 3 + 4 é representada pelo seguinte objeto **`Exp`**:

```
Exp e1 = new Soma(
```

```
                new Produto(
                  new Num(5),
                  new Num(3)
                ),
               new Num(4)
  )
```

O cliente, elemento que manipula os objetos na composição através da interface **`Exp`**, usa a interface da classe para interagir com os objetos na estrutura composta. Se o receptor é um objeto **`Num`** ou **`Racional`**, então a solicitação é tratada diretamente. Se o receptor é um objeto **`Soma`** ou **`Produto`**, então parte da computação envolvida é delegada aos seus componentes-filhos.

Como conseqüência do uso deste padrão percebemos a definição das hierarquias de classe que consistem de objetos primitivos e de objetos compostos. Os objetos primitivos podem compor objetos mais complexos, os quais, por sua vez, também podem compor outros objetos, e assim por diante, recursivamente. O código do cliente pode aceitar um objeto primitivo ou um objeto composto.

O comportamento do cliente torna-se simples mesmo em face de estruturas composicionais complexas. Isso simplifica o código da classe cliente e evita o uso de comandos *case* que utilizam os rótulos das classes que definem a composição [GAM2000].

Essa arquitetura torna mais fácil o acréscimo de novos tipo de componentes às expressões que o avaliador pode tratar. Os clientes não precisam ser modificados para tratar novas classes do tipo **`Exp`**.

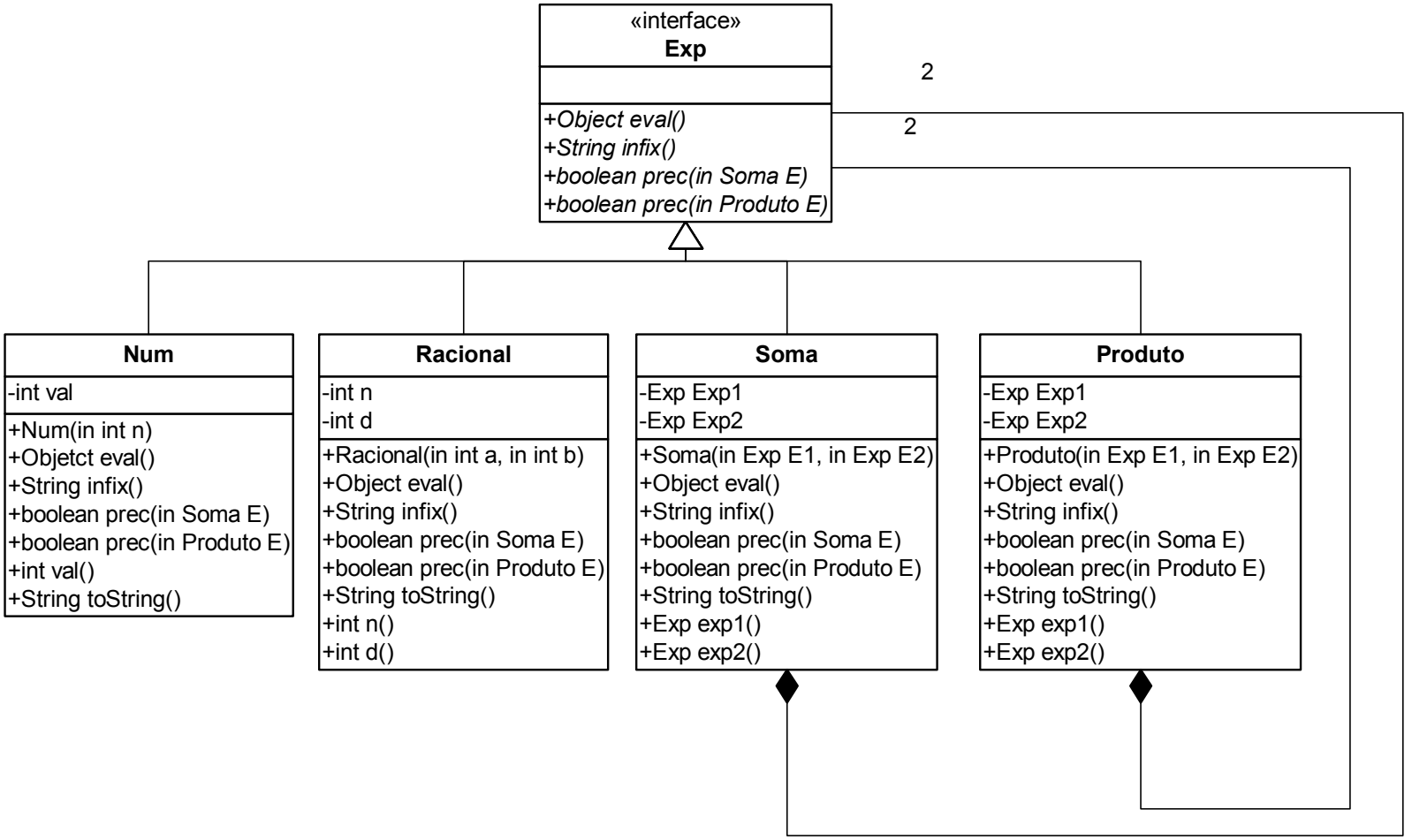


Figura 4.4 Diagrama de classes da versão 0.1.

A Fig. 4.4 apresenta o projeto completo da versão 0.1. Como podemos ver, a interface **`Exp`** especifica quatro métodos. O Método

```
public Object eval() { ... }
```

retorna um objeto que corresponde ao valor da expressão. O comportamento associado a este método opera basicamente por delegação. Por exemplo, ao encontrar um objeto Soma, ele delega aos seus dois operandos, referências **`Exp1`** e **`Exp2`**, o cálculo de seus valores, digamos que seja **`15`** e **`4`**. Só então retorna a resposta que é um obejto **`new Num(19)`**.

Os três outros métodos são relacionados a representação concreta das expressões. O método

```
public String infix(){ ... }
```

fornece uma representação de objetos **`Exp`** em termos de seqüência de caracteres onde as operações binárias são representadas na forma sintática infixada – operandos entre o operador.

Os outros dois métodos

```
public boolean prec(Soma E) { ... }
public boolean prec(Produto E) { ... }
```

fornecem a informação sobre a precedência entre os operadores binários. Por exemplo, uma mensagem **`prec(x)`** para um objeto **`Produto`** responderá **`true`** sempre que **`x`** for um objeto **`Soma`**, pois a multiplicação tem precedência sobre a adição. Nessa abordagem, é oferecida apenas a precedência entre os operadores binários. Esse mesmo esquema poderá ser estendido a outros tipos de operação.

O projeto apresentado na Fig. 4.4 é totalmente funcional e já exibe boas características de reusabilidade. Por exemplo poderíamos acrescentar operações como subtração, divisão e exponenciação com relativamente poucas mudanças no projeto e no código das classes. Entretanto, a inclusão de mais um tipo de avaliação, por exemplo, uma que retorne o valor correto da expressão 2 + 4/3 que é o racional 10/3, exigirá modificações no código de todas as classes que realizam a interface Exp. Certamente um sério problema de reusabilidade. A próxima seção dedica-se ao estudo desse problema e dá origem a versão 1.0.

4 - Tratando o problema da avaliação de expressões – versão 1.0

Como vimos na seção anterior, a inclusão de novos tipos de avaliação de expressões exigirá alterações no código das classes realizando a interface **`Exp`**. Na versão 1.0 do avaliador, foco principal desta seção, este problema é resolvido com a utilização do padrão Visitante [GAM2002]. Ele fornecerá o descolamento entre os dados e o comportamento associados às avaliações. Os dados continuaram a residir nos próprios objetos Exp, mas o comportamento estará associado a um outro tipo de objeto, os *avaliadores*. Antes de prosseguir com os detalhes desta solução, uma palavrinha sobre este padrão.

O padrão Visitante tem por objetivo a representação de operações a serem executadas nos elementos de uma estrutura de objetos. Permite definir uma nova operação sem modificar as classes dos objetos sobre as quais opera. É utilizado quando uma estrutura de objetos envolve muitas interface e desejamos executar operações que dependem de suas classes concretas.

Outra situação de aplicação é quando muitas operações distintas e não-relacionadas necessitam ser executadas sobre objetos de uma estrutura de objetos. Neste caso, o padrão Visitante é utilizado para manter coesas operações relacionadas entre si, definindo-as em uma única classe.

Uma terceira situação de uso do padrão é quando as classes que definem a estrutura do objeto raramente mudam, porém, freqüentemente desejamos definir novas operações sobre a estrutura. A mudança das classes da estrutura do objeto requer a redefinição da interface para todos os visitantes, o que pode vir a se tornar oneroso. Se as classes da estrutura do objeto mudam com freqüência, então é provavelmente melhor definir as operações nas próprias classes.

Foi necessário fazer uma alteração na estrutura da interface **`Exp`** (cf. Fig. 4.5), que agora passa a contar com um método que aceita um objeto visitante como argumento – objetos EvalVisitante.

```
interface Exp {
  Object aceita(EvalVisitante V);
  String infix();
  boolean prec(Soma E);
  boolean prec(Produto E);
}
```

Figura 4.5 Código da interface Exp na versão 1.0.

Diferente da versão 0.1, a avaliação da expressão não está a cargo do próprio objeto **`Exp`**. O processo passa a ser feito por um objeto visitante que realiza a interface **`EvalVisitante`** (cf. Fig. 4.6). O visitante recebe os elementos da expressão à medida que ela vai sendo percorrida. O comportamento da avaliação é totalmente encapsulado pelos objetos **`EvalVisitante`**, desonerando os objetos **`Exp`** dessa tarefa. Portanto, desacoplando as classes relacionadas aos dados, as classes **`Exp`**, das classes relacionadas às operações, as classes **`EvalVisitante`**.

```
interface EvalVisitante {
  public Object visitaNum(int val);
  public Object visitaRacional(int n, int d);
  public Object visitaSoma(Exp E1, Exp E2);
  public Object visitaProduto(Exp E1, Exp E2);
}
```

Figura 4.6 Código da interface EvalVisitante na versão 1.0.

A interface **`EvalVisitante`** deve declarar uma operação para cada tipo de expressão para que possam ser adotados diferentes comportamentos de avaliação de expressão, conforme o tipo repassado no momento.

O padrão Visitante encapsula as operações para cada fase do processo de avaliação da expressão em uma classe que realiza a interface **`EvalVisitante`** associada a fase.

O diagrama de classes dessa versão está representado na Fig. 4.7 e traz como principal alteração em relação à versão anterior a criação de uma novo conjunto de classes, baseadas na interface **`EvalVisitante`**.

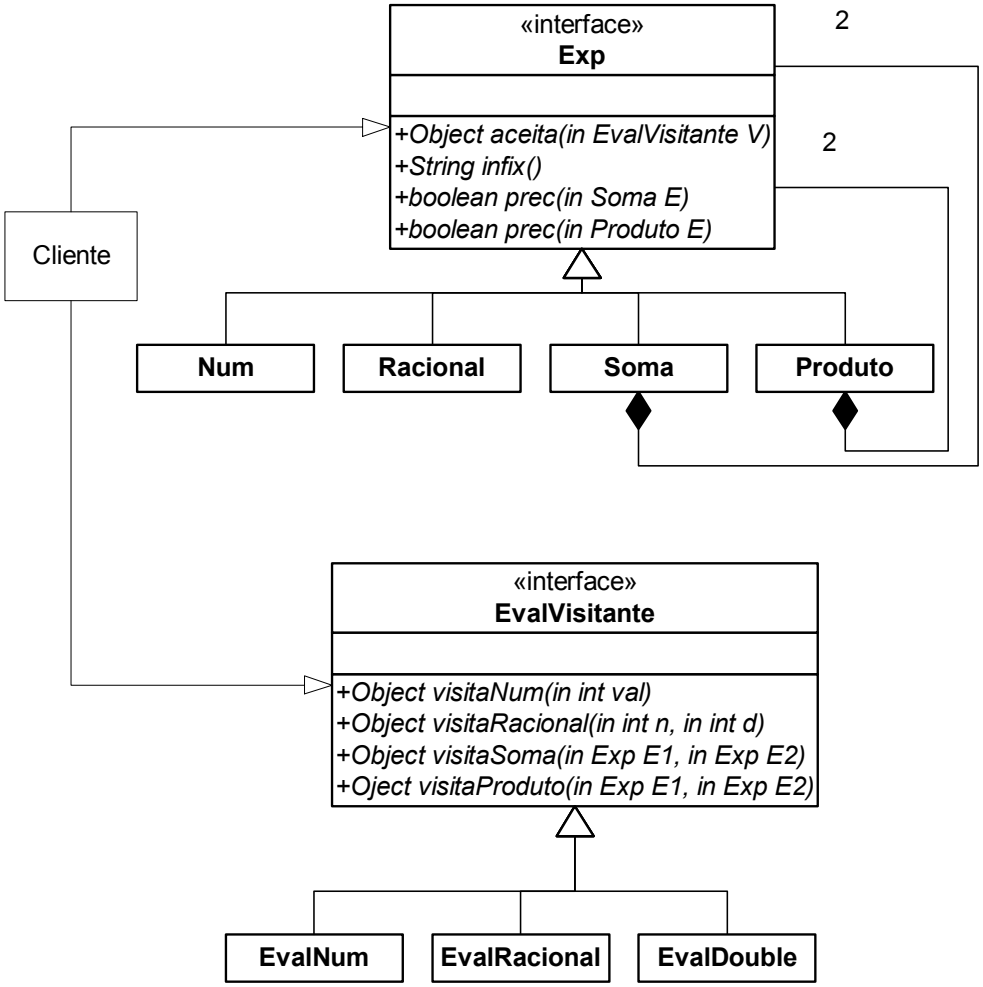


Figura 4.7 Diagrama de classes da versão 1.0.

Esse descolamento entre os dados e o comportamento no processo de avaliação de uma expressão permite que novos tipos de avaliadores possam ser inseridos no projeto sem *nenhuma alteração* do código das classes que realizam a interface **`Exp`**. De fato, o projeto permite uma estrutura de aplicação totalmente orientada a componentes para a funcionalidade relacionada à avaliação de expressões. Essa característica não é, no entanto, contemplada no protótipo desenvolvido nessa dissertação.

A aplicação do padrão Visitante permite que sejam definidas duas hierarquias de classes: uma para os elementos sobre os quais estão sendo aplicadas as operações (a hierarquia **`Exp`**) e uma para os visitantes que definem as operações sobre os elementos (a hierarquia **`EvalVisitante`**). Com essa solução, podemos criar novas operações e funcionalidades acrescentando uma nova subclasse à hierarquia da classe **`EvalVisitante`**. O usuário pode avaliar a expressão 2 + 4/3 e obter como resultado o inteiro 3, o racional 10/3 ou o decimal 3,33 dependendo do objeto **`EvalVisitante`** escolhido.

O projeto do avaliador apresentado na Fig. 4.7 ainda possui um inconveniente importante. A adição novos tipos de expressão esbarra com o problema da representação concreta das expressões, pois precisa conhecer a precedência entre todos os tipos de expressão. A próxima seção examina este problema e propõe uma solução baseada no padrão Método de Fábrica.

5 - Revendo o problema da representação concreta de expressões – versão 1.1

Vimos na seção **4.4** que a representação concreta dependia da precedência entre os diversos tipos de operadores binários. A solução encontrada até então utiliza a informação sobre a precedência fornecida pelos próprios objetos **`Exp`** através de métodos fornecendo a precedência de cada tipo de operação em relação aos demais – os métodos **`prec()`**.

O problema reside especificamente no fato de que a realização dos métodos utiliza uma informação que é de âmbito global do projeto: a precedência entre todos os

tipos de expressão. À medida que um novo tipo de expressão é adicionado ao projeto, é necessária uma atualização nos demais. Para postergar o conhecimento desse tipo de informação, associa-se a cada tipo de expressão um novo tipo de objeto – **`Precedencia`** – que encapsula sobre si a informação sobre a precedência (cf. Fig. 4.8). Ao desobrigar objetos **`Exp`** da responsabilidade de conhecerem a sua precedência em relação aos demais, eliminamos o acoplamento entre os diferentes tipos de expressões, e conseqüentemente, aumentamos o grau de reusabilidade do projeto.

```
interface Precedencia {
  boolean prec(Soma E);
  boolean prec(Produto E);
}
```

Figura 4.8 Código da interface Precedencia na versão 1.1.

A solução adotada na versão 1.1 utiliza o padrão Método Fabrica. Nesta versão, cada objeto **`Exp`** acessa o seu objeto **`Precedencia`** a partir do método **`mkPrecedencia`** – cf. Fig. 4.9.

```
interface Exp {
  Object aceita(EvalVisitante V);
  String infix();
  Precedencia mkPrecedencia();
}
```

Figura 4.9 Código da interface Exp na versão 1.1.

O padrão Método de Fábrica é geralmente utilizado postergar a criação de um objeto para o momento de sua utilização. Quando uma classe não pode antecipar a classe de objetos que deve criar ou quando quisermos que subclasses de uma classe especifiquem os objetos que criam. Uma outra situação propícia ao uso é quando classes delegam responsabilidade para uma entre várias subclasses auxiliares e é necessário conhecer qual delas será delegada. No nosso caso, cada objeto **`Exp`** delega aos seus componentes (operandos) a criação dos objetos **`Precedencia`**.

A nova organização da estrutura de interfaces e classes é mostrada pela Fig. 4.10.

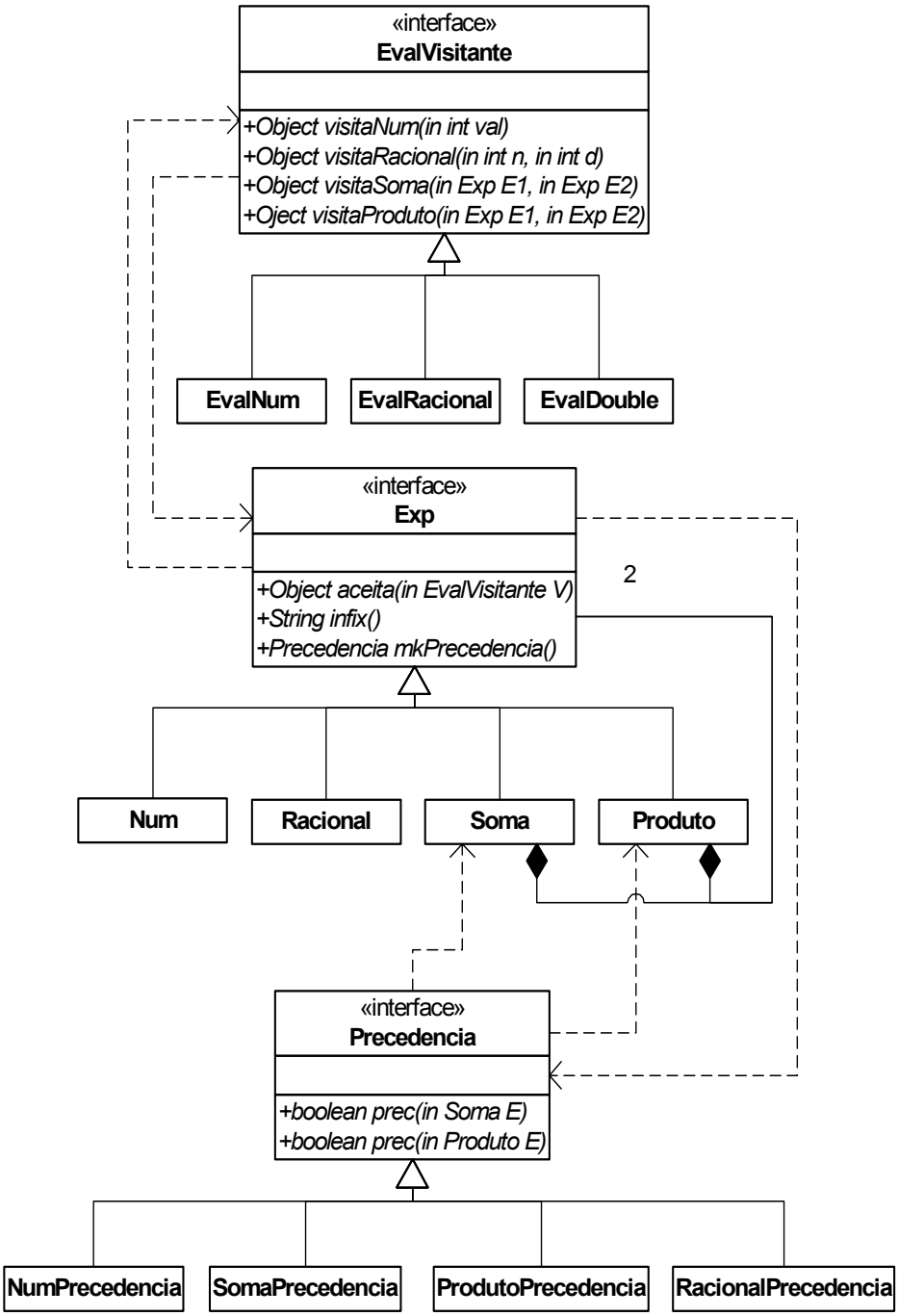


Figura 4.10 Diagrama de classes da solução 1.1.

As classes realizam a interface **`Exp`** redefinem a operação **`mkPrecedencia`** para retornar a subclasse **`Precedencia`** apropriada. Uma vez que uma classe do tipo Exp é instanciada, o objeto **`Precedencia`** associado ao tipo de expressão que a instância representa pode ser instanciado sem o conhecimento especifico de outras classes Exp. O objeto **`Precedencia`** retornado pelo método **`mkPrecedencia`** ocupa-se desta parte. O método **`mkPrecedencia`** é considerado Método de Fábrica porque ele é responsável pela manufatura de um objeto.

A solução adotada pela versão 1.1 retira dos objetos **`Exp`** o conhecimento sobre a precedência das operações, mas ainda os responsabiliza pela criação dos objetos **`Precedencia`**. Portanto, mantém em algum grau a necessidade de conhecimento mútuo entre as classes que realizam a interface **`Exp`**. A próxima seção corrige esse inconveniente a partir do uso do padrão Fábrica Abstrata.

6 - Ainda sobre o problema da representação concreta de expressões – versão 1.2

Nesta versão do projeto, cada objeto **`Exp`** acessa o seu objeto **`Precedência`** a partir do método **`mkPrecedencia`** que recebe como argumento um objeto do tipo **`PrecedenciaAbsFab`** – cf. Fig. 4.12. Esse argumento é conhecido pelo objeto **`Exp`** no momento da invocação do método **`infix`** – cf Fig. 4.11. É o uso conjunto de dois padrões de projeto: o padrão Método de Fábrica, utilizado na versão precedente do projeto, e o padrão Fábrica Abstrata.

```
interface Exp {
  Object aceita(EvalVisitante V);
  String infix(PrecedenciaAbsFab f);
  Precedencia mkPrecedencia(PrecedenciaAbsFab f);
}
```

Figura 4.11 Código da interface Exp na versão 1.2.

```
interface PrecedenciaAbsFab {
  Precedencia mkNumPrecedencia();
  Precedencia mkSomaPrecedencia();
  Precedencia mkProdutoPrecedencia();
  Precedencia mkRacionalPrecedencia();
}
```

Figura 4.12 Código da interface PrecedenciaAbsFab na versão 1.2.

O padrão Fábrica Abstrata oferece uma interface para a criação de objetos que serão conhecidos mais tarde no projeto. Esse papel é desempenhado pela interface **`PrecedenciaAbsFab`** para o caso dos nossos objetos **`Precedencia`**. O projeto na sua versão 1.2 é apresentado pela Fig. 4.13.

O padrão Fábrica Abstrata é geralmente utilizado nas seguintes situações [GAM2000]:

- Quando um sistema deve ser independente da maneira pela qual os seus produtos são criados, compostos ou representados;
- Quando um sistema deve ser configurado como um produto de uma família de múltiplos produtos;
- Quando uma família de objetos for projetada para ser utilizada em conjunto, sendo necessário garantir essa restrição;
- Quando for preciso fornecer uma biblioteca de classes de produtos e revelar somente suas interfaces, não suas realizações.

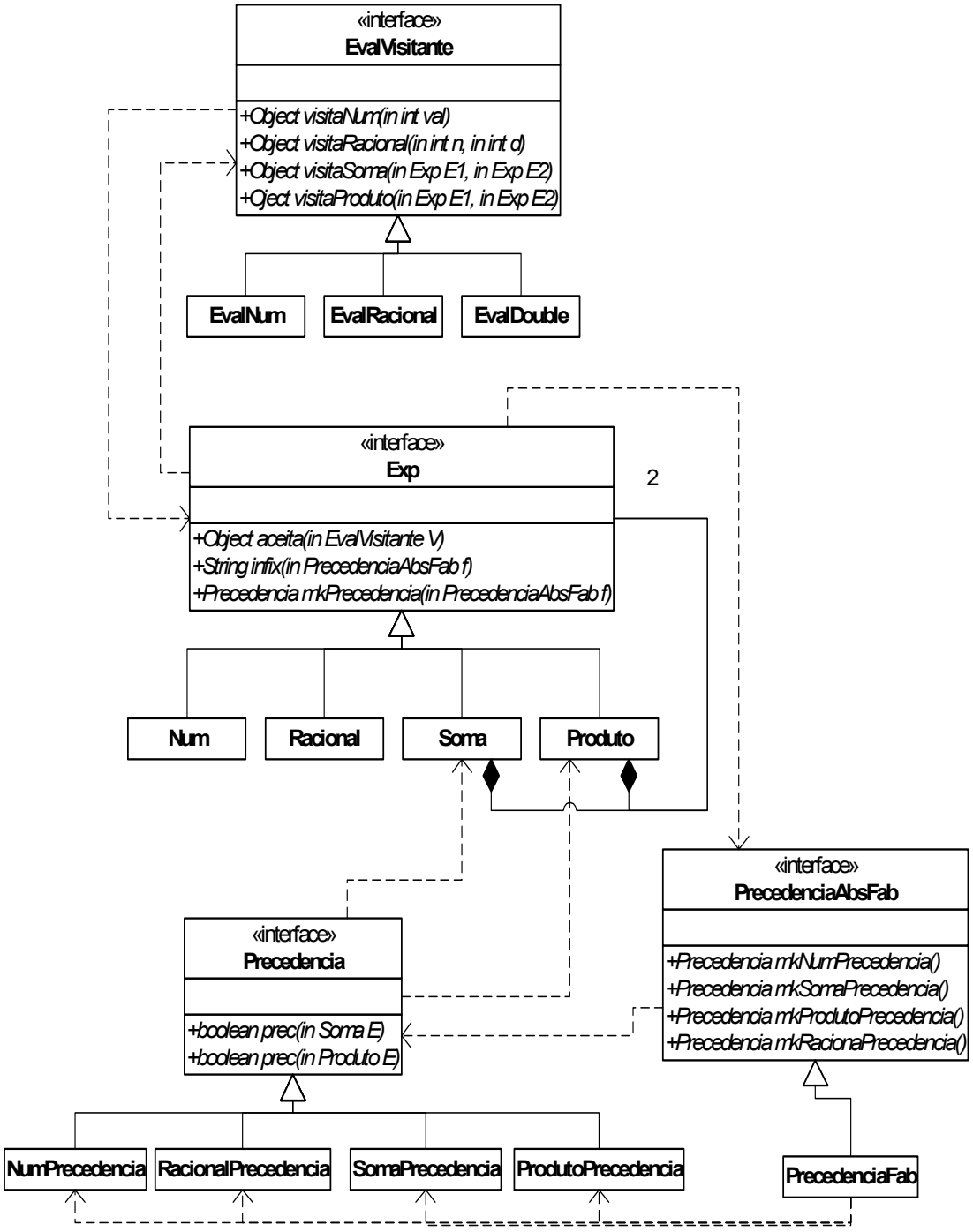


Figura 4.13 Diagrama de classes da solução 1.2.

O padrão Fábrica Abstrata isola as classes concretas, pois ajuda a controlar as classes de objetos criadas por uma aplicação. Uma vez que a fábrica encapsula a responsabilidade e o processo de criar objetos, isola os clientes das classes de realização. Os clientes manipulam as instâncias através das suas interfaces abstratas.

Nomes de classes ficam isolados na realização da fábrica concreta; eles não aparecem no código do cliente.

Desonerando os objetos **`Exp`** da responsabilidade da criação dos objetos **`Precedencia`** tornamos as suas especificações independentes. Novos tipos de expressão poderão ser independentemente adicionados ao projeto. O conhecimento sobre a precedência das operações é inteiramente encapsulado pelo objeto **`PrecedenciaAbsFab`** passado como parâmetro na invocação do método **`infix()`**. Apenas a classe fornecendo a realização da interface **`PrecedenciaAbsFab`** deve ser atualizada. Os códigos das classes **`Exp`** tornam-se totalmente independentes entre si.

7 - Conclusão

A aplicação do padrão Composto, na versão 0.1, possibilitou a agregação dos elementos da expressão em uma estrutura de árvore, caracterizando a natureza recursiva de uma expressão matemática, distribuindo os elementos em nós e folhas e no momento em que se chega ao valor da raiz, consegue-se o valor da expressão.

O uso do padrão Visitante na versão 1.0 possibilitou que a responsabilidade de avaliar a expressão ficasse a cargo de um cliente visitante, e não mais para a própria classe realizadora de **`Exp`**. Isso permitiu que fossem elaborados diferentes métodos de cálculo tornando flexível o processo de avaliação de uma expressão.

Os padrões Método de Fábrica e Fábrica Abstrata foram conjuntamente utilizados nas versões 1.1 e 1.2 para fornecer o desacoplamento entre os próprios objetos **`Exp`** de forma a permitir a inclusão de novos tipos de expressão com um mínimo de modificação nos códigos.

De uma forma geral a utilização de padrões no avaliador de expressões melhorou a estrutura, facilitando o entendimento da solução, reusabilidade e relação entre seus elementos. Especificamente para o caso da avaliação de expressões, foi possível obter um projeto totalmente componentizável.

Capítulo 5

Conclusão

A complexidade dos sistemas de software tem aumentado tanto que a utilização de técnicas e conceitos da orientação a objetos e a representação de seus modelos de software através de uma linguagem padronizada tornou-se imprescindível. É neste contexto que os padrões de projeto se aplicam. Proporcionando o reuso de técnicas e soluções já conhecidas para resolver problemas recorrentes, os padrões de projeto vêm sendo cada vez mais considerados ferramentas essenciais para o projeto reutilizável de software.

Neste trabalho de dissertação propôs-se um estudo no qual os padrões foram seqüencialmente utilizados para prover soluções para o projeto reutilizável de um avaliador de expressões matemáticas. Além de investigar o uso dessa técnica de projeto de software através de um estudo de caso, o estudo buscou evidenciar a inevitabilidade entre o uso de padrões e o projeto reutilizável de software. A idéia é que padrões são mecanismos que permitem a abstração de elementos arquiteturais de software. Isso ficou patente no estudo de caso apresentado no Cap. 4 na solução de diversos problemas de projeto.

Os próximos parágrafos discutem algumas propostas de continuação deste trabalho de dissertação.

O software descrito no Cap. 4 embora já bastante estruturado peca por acabamento. Seria obviamente interessante poder contar com um ambiente visual, mesmo que simples, oferecendo, entre outras funcionalidades, uma interface amigável para a escrita das expressões.

Um dos ápices da reusabilidade é o uso de componentes. Ficou bastante claro no projeto apresentado no Cap. 4 que os diferentes tipo de avaliações de expressões poderiam ser facilmente manipulados na forma de componentes. Com um pouco mais de esforço, provavelmente poderemos também tratar outros elementos desse projeto como componentes. O estudo e a proposição de um projeto totalmente orientado

a componentes para o avaliador de expressões é colocado como uma outra proposta de continuação desse trabalho.

Este trabalho de dissertação abordou o aspecto da utilização de padrões de projeto compromissada quase que exclusivamente ao aspecto de reusabilidade. Outros aspectos da utilização de padrões são também importantes no projeto de sistemas. Em particular, a aplicação de padrões geralmente afeta o desempenho do sistema negativamente. Como uma terceira proposta de continuação, técnicas de otimização de código, por exemplo, poderão ser estudadas para estabelecer um balanço adequado entre os bônus e ônus da utilização de padrões.

Referências

[BAC2000] BACHMAN, Felix; BASS, Len; BUHMAN, Charles et al. Technical Concepts of Component-Based Software Engineering. CMU/SEI 2000-TR 008. Volume II. May 2000.

[BOO1994] BOOCH, Grady. Object-Oriented Analysis and Design: With Applications. Addison Wesley, 1994. Second Edition.

[BOO1998a] BOOCH, Grady. Objected-Oriented Analysis and Design with applications. Addison-Wesley, 1998.

[BOO1998b] BOOCH, Grady; KOZACZYNSKI, Wojtek. Component-Based Software Engineering. IEEE Software. Setembro/Outubro 1998.

[BOO2000] BOOCH, Grady; JACOBSON, Ivar; RUMBAUGH, James. UML: Guia do Usuário. Rio de Janeiro: Campus, 2000.

[BRO1998] BROWN, Alan; WALLNAU, C. Kurt. The current State of CBSE. IEEE Software. Setembro/Outubro 1998.

[BRO2000] BROWN, Alan W. Large-Scale, Component-Based Development. Prentice-Hall 2000.

[CES1996] CESTA, André Augusto. Tutorial: A Linguagem de Programação Java. Unicamp, 1996.

[COA1998] COAD, Peter; MAYFIELD, Mark. Projeto de Sistemas em Java: Construindo aplicativos e melhores applets. São Paulo, Makron Books, 1998.

[DEI2001] DEITEL, H. M.; DEITEL, P. J.Java, como programar. 3ª edição. Porto Alegre: Bookman, 2001.

[DIG1998] DIGRE, Tom. Bussiness Object Componente Architecture. IEEE Software. Setembro/Outubro 1998.

[ECK2002] ECKEL, Bruce. Thinking in Patterns with Java. MindView Inc. Revision 0.4. Livro Eletrônico obtido em Julho/2002.

[FOW2000] FOWLER, Martin; SCOTT, Kendal. UML essencial: um breve guia para a linguagem padrão de modelagem de objetos. Porto Alegre, Bookman, 2000.

[FOW2002] FOWLER, Martin. Enterprise Application Architecture. Obtido da Internet no endereço http://martinfowler.com/isa/index.html em 30/01/2002.

[GAM2000] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos. Porto Alegre: Bookman, 2000.

[LAR2000] LARMAN, Craig. Utilizando UML e Padrões: uma introdução à análise e ao projeto orientados a objetos. Porto Alegre: Bookman, 2000.

[LEW1998] LEWIS, John; LOFTUS, William. Java: Software Solutions. Addison-Wesley, 1998.

[MAT2002] MATOS, Alexandre Veloso de. UML Prático e Descomplicado. São Paulo: Érica, 2002.

[OLI1996] OLIVEIRA, Adelize Generini de. Análise, Projeto e Programação Orientados a Objetos.Bookstore, 1996.

[RAM2001] RAMACHANDRAN, Vijay S. Design Patterns for Optimizing the Performance of J2EE Applications. Dezembro/2001. Obtido na Internet no endereço http:// developer.java.sun.com/ developer/ technicalArticles/ J2EE/ J2EEpatterns/ em 22/02/2002.

[ROW1998] ROWE, Glenn. An Introduction to Data Structures and Algorithms with Java. Prentice hall Europe, 1998.

[SCH1998] SCHNEIDER, Geri;. WINTERS, Jason P. Applying Use Cases: A practical guide. Addison-Wesley, 1998.

[SUN2002] Sun Microsystems. Model-View-Controller Architecture Sun Microsystems obtido na Internet no endereço http:// java.sun.com/ blueprints/ patterns/ j2ee_patterns/ model_view_controller/ index.html em 10/07/2002.